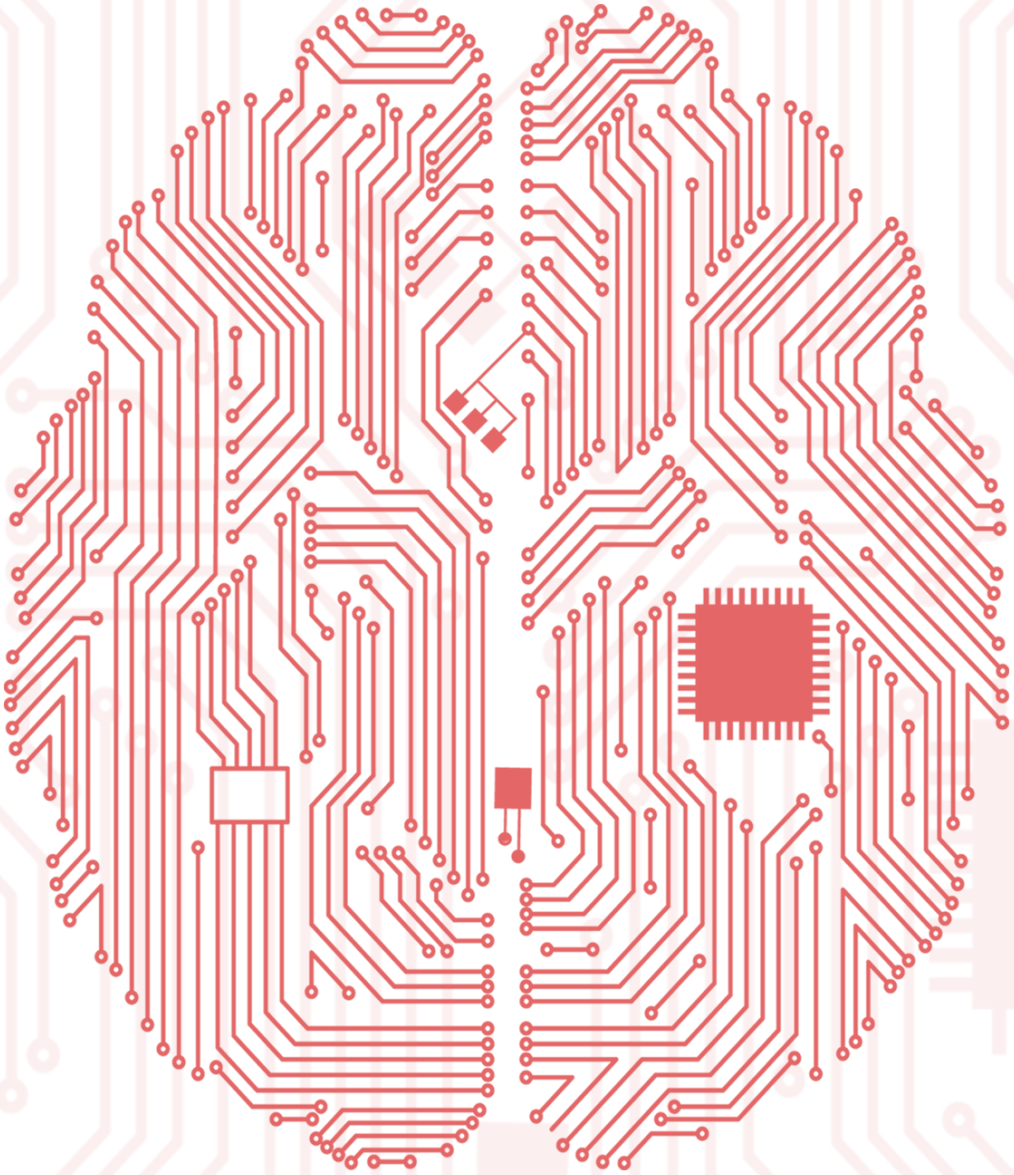
DEEP LEARNING FOR COMPUTER VISION



WITH PYTHON

Dr. Adrian Rosebrock

pyimagesearch

Deep Learning for Computer Vision with Python

Practitioner Bundle

Dr. Adrian Rosebrock

1st Edition (1.2.1)

Copyright © 2017 Adrian Rosebrock, PyImageSearch.com

PUBLISHED BY PYIMAGESEARCH

PYIMAGESEARCH.COM

The contents of this book, unless otherwise indicated, are Copyright ©2017 Adrian Rosebrock, PyimageSearch.com. All rights reserved. Books like this are made possible by the time invested by the authors. If you received this book and did not purchase it, please consider making future books possible by buying a copy at https://www.pyimagesearch.com/deep-learning-computer-vision-python-book/ today.

First printing, September 2017

To my father, Joe; my wife, Trisha;
and the family beagles, Josie and Jemma.
Without their constant love and support,
this book would not be possible.

Contents

1 Introduction 13

2 Introduction 15

3 Training Networks Using Multiple GPUs 17
3.1 How Many GPUs Do I Need? 17
3.2 Performance Gains Using Multiple GPUs 18
3.3 Summary 19

4 What Is ImageNet? 21
4.1 The ImageNet Dataset 21
4.1.1 ILSVRC 21
4.2 Obtaining ImageNet 23
4.2.1 Requesting Access to the ILSVRC Challenge 23
4.2.2 Downloading Images Programmatically 23
4.2.3 Using External Services 24
4.2.4 ImageNet Development Kit 24
4.2.5 ImageNet Copyright Concerns 25
4.3 Summary 27

5 Preparing the ImageNet Dataset 29
5.1 Understanding the ImageNet File Structure 29
5.1.1 ImageNet "test" Directory 30
5.1.2 ImageNet "train" Directory 31
5.1.3 ImageNet "val" Directory 32
5.1.4 ImageNet "ImageSets" Directory 33

5.1.5 ImageNet "DevKit" Directory 34
5.2 Building the ImageNet Dataset 37
5.2.1 Your First ImageNet Configuration File 37
5.2.2 Our ImageNet Helper Utility 42
5.2.3 Creating List and Mean Files 46
5.2.4 Building the Compact Record Files 50
5.3 Summary 52

6 Training AlexNet on ImageNet 53
6.1 Implementing AlexNet 54
6.2 Training AlexNet 58
6.2.1 What About Training Plots? 59
6.2.2 Implementing the Training Script 60
6.3 Evaluating AlexNet 65
6.4 AlexNet Experiments 67
6.4.1 AlexNet: Experiment #1 68
6.4.2 AlexNet: Experiment #2 70
6.4.3 AlexNet: Experiment #3 71
6.5 Summary 74

7 Training VGGNet on ImageNet 75
7.1 Implementing VGGNet 76
7.2 Training VGGNet 81
7.3 Evaluating VGGNet 85
7.4 VGGNet Experiments 86
7.5 Summary 88

8 Training GoogLeNet on ImageNet 89
8.1 Understanding GoogLeNet 89
8.1.1 The Inception Module 90
8.1.2 GoogLeNet Architecture 90
8.1.3 Implementing GoogLeNet 91
8.1.4 Training GoogLeNet 95
8.2 Evaluating GoogLeNet 99
8.3 GoogLeNet Experiments 99
8.3.1 GoogLeNet: Experiment #1 100
8.3.2 GoogLeNet: Experiment #2 101
8.3.3 GoogLeNet: Experiment #3 102
8.4 Summary 103

9 Training ResNet on ImageNet 105
9.1 Understanding ResNet 105
9.2 Implementing ResNet 106
9.3 Training ResNet 112
9.4 Evaluating ResNet 116

9.5 ResNet Experiments 116
9.5.1 ResNet: Experiment #1 . . . 116
9.5.2 ResNet: Experiment #2 . . . 116
9.5.3 ResNet: Experiment #3 . . . 117
9.6 Summary 120

10 Training SqueezeNet on ImageNet . . . 121
10.1 Understanding SqueezeNet 121
10.1.1 The Fire Module . . . 121
10.1.2 SqueezeNet Architecture . . . 123
10.1.3 Implementing SqueezeNet . . . 124
10.2 Training SqueezeNet 128
10.3 Evaluating SqueezeNet 132
10.4 SqueezeNet Experiments 132
10.4.1 SqueezeNet: Experiment #1 . . . 132
10.4.2 SqueezeNet: Experiment #2 . . . 134
10.4.3 SqueezeNet: Experiment #3 . . . 135
10.4.4 SqueezeNet: Experiment #4 . . . 136
10.5 Summary 139

11 Case Study: Emotion Recognition . . . 141
11.1 The Kaggle Facial Expression Recognition Challenge 141
11.1.1 The FER13 Dataset . . . 141
11.1.2 Building the FER13 Dataset . . . 142
11.2 Implementing a VGG-like Network 147
11.3 Training Our Facial Expression Recognizer 150
11.3.1 EmotionVGGNet: Experiment #1 . . . 153
11.3.2 EmotionVGGNet: Experiment #2 . . . 153
11.3.3 EmotionVGGNet: Experiment #3 . . . 154
11.3.4 EmotionVGGNet: Experiment #4 . . . 155
11.4 Evaluating our Facial Expression Recognizer 157
11.5 Emotion Detection in Real-time 159
11.6 Summary 163

12 Case Study: Correcting Image Orientation . . . 165
12.1 The Indoor CVPR Dataset 165
12.1.1 Building the Dataset . . . 166
12.2 Extracting Features 170
12.3 Training an Orientation Correction Classifier 173
12.4 Correcting Orientation 175
12.5 Summary 177

13 Case Study: Vehicle Identification . . . 179
13.1 The Stanford Cars Dataset 179
13.1.1 Building the Stanford Cars Dataset . . . 180

13.2 Fine-tuning VGG on the Stanford Cars Dataset 187
13.2.1 VGG Fine-tuning: Experiment #1 . . . 192
13.2.2 VGG Fine-tuning: Experiment #2 . . . 193
13.2.3 VGG Fine-tuning: Experiment #3 . . . 194
13.3 Evaluating our Vehicle Classifier 195
13.4 Visualizing Vehicle Classification Results 197
13.5 Summary 201

14 Case Study: Age and Gender Prediction . . . 203
14.1 The Ethics of Gender Identification in Machine Learning 203
14.2 The Adience Dataset 204
14.2.1 Building the Adience Dataset . . . 205
14.3 Implementing Our Network Architecture 219
14.4 Measuring "One-off" Accuracy 221
14.5 Training Our Age and Gender Predictor 224
14.6 Evaluating Age and Gender Prediction 227
14.7 Age and Gender Prediction Results 230
14.7.1 Age Results . . . 230
14.7.2 Gender Results . . . 231
14.8 Visualizing Results 233
14.8.1 Visualizing Results from Inside Adience . . . 234
14.8.2 Understanding Face Alignment . . . 238
14.8.3 Applying Age and Gender Prediction to Your Own Images . . . 240
14.9 Summary 244

15 Faster R-CNNs . . . 247
15.1 Object Detection and Deep Learning 247
15.1.1 Measuring Object Detector Performance . . . 248
15.2 The (Faster) R-CNN Architecture 250
15.2.1 A Brief History of R-CNN . . . 250
15.2.2 The Base Network . . . 254
15.2.3 Anchors . . . 255
15.2.4 Region Proposal Network (RPN) . . . 257
15.2.5 Region of Interest (ROI) Pooling . . . 258
15.2.6 Region-based Convolutional Neural Network . . . 259
15.2.7 The Complete Training Pipeline . . . 260
15.3 Summary 260

16 Training a Faster R-CNN From Scratch . . . 261
16.1 The LISA Traffic Signs Dataset 261
16.2 Installing the TensorFlow Object Detection API 262
16.3 Training Your Faster R-CNN 263
16.3.1 Project Directory Structure . . . 263
16.3.2 Configuration . . . 265
16.3.3 A TensorFlow Annotation Class . . . 267

16.3.4 Building the LISA + TensorFlow Dataset 269
16.3.5 A Critical Pre-Training Step 274
16.3.6 Configuring the Faster R-CNN 275
16.3.7 Training the Faster R-CNN 280
16.3.8 Suggestions When Working with the TFOD API 282
16.3.9 Exporting the Frozen Model Graph 286
16.3.10 Faster R-CNN on Images and Videos 286
16.4 Summary **290**

17 Single Shot Detectors (SSDs) **293**
17.1 Understanding Single Shot Detectors (SSDs) **293**
17.1.1 Motivation 293
17.1.2 Architecture 294
17.1.3 MultiBox, Priors, and Fixed Priors 295
17.1.4 Training Methods 296
17.2 Summary **297**

18 Training a SSD From Scratch **299**
18.1 The Vehicle Dataset **299**
18.2 Training Your SSD **300**
18.2.1 Directory Structure and Configuration 300
18.2.2 Building the Vehicle Dataset 302
18.2.3 Training the SSD 307
18.2.4 SSD Results 310
18.2.5 Potential Problems and Limitations 311
18.3 Summary **312**

19 Conclusions **313**
19.1 Where to Now? **314**

Companion Website

Thank you for picking up a copy of *Deep Learning for Computer Vision with Python!* To accompany this book I have created a companion website which includes:

- **Up-to-date installation instructions** on how to configure your development environment
- Instructions on how to use the **pre-configured Ubuntu VirtualBox virtual machine** and **Amazon Machine Image (AMI)**
- **Supplementary material** that I could not fit inside this book
- **Frequently Asked Questions (FAQs)** and their suggested fixes and solutions

Additionally, you can use the "Issues" feature inside the companion website to report any bugs, typos, or problems you encounter when working through the book. I don't expect many problems; however, this is a brand new book so myself and other readers would appreciate reporting any issues you run into. From there, I can keep the book updated and bug free.

To create your companion website account, just use this link:

http://pyimg.co/fnkxk

Take a second to create your account now so you'll have access to the supplementary materials as you work through the book.

1. Introduction

Welcome to the *Practitioner Bundle* of *Deep Learning for Computer Vision with Python!* This volume is meant to be the next logical step in your deep learning for computer vision education after completing the *Starter Bundle*.

At this point, you should have a strong understanding of the *fundamentals* of parameterized learning, neural networks, and Convolutional Neural Networks (CNNs). You should also feel relatively comfortable using the Keras library and the Python programming language to train your own custom deep learning networks.

The purpose of the *Practitioner Bundle* is to build on your knowledge gained from the *Starter Bundle* and introduce more advanced algorithms, concepts, and tricks of the trade — these techniques will be covered in three distinct parts of the book.

The first part will focus on methods that are used to boost your classification accuracy in one way or another. One way to increase your classification accuracy is to apply transfer learning methods such as fine-tuning or treating your network as a feature extractor.

We'll also explore *ensemble methods* (i.e., training *multiple networks* and combining the results) and how these methods can give you a nice classification boost with little extra effort. Regularization methods such as *data augmentation* are used to generate additional training data – in nearly all situations, data augmentation improves your model's ability to generalize. More advanced optimization algorithms such as Adam [1], RMSprop [2], and others can also be used on some datasets to help you obtain lower loss. After we review these techniques, we'll look at the *optimal pathway to apply these methods* to ensure you obtain the maximum amount of benefit with the least amount of effort.

We then move on to the second part of the *Practitioner Bundle* which focuses on *larger datasets* and *more exotic network architectures*. Thus far we have only worked with datasets that have fit into the main memory of our system – but what if our dataset is too large to fit into RAM? What do we do then? We'll address this question in Chapter **??** when we work with HDF5.

Given that we'll be working with larger datasets, we'll also be able to discuss more advanced network architectures using AlexNet, GoogLeNet, ResNet, and deeper variants of VGGNet. These network architectures will be applied to more challenging datasets and competitions, including the

Kaggle: Dogs vs. Cats recognition challenge [3] as well as the cs231n Tiny ImageNet challenge [4], the exact same task Stanford CNN students compete in. As we'll find out, we'll be able to obtain a top-25 position on the Kaggle Dogs vs. Cats leaderboard and top the cs231n challenge for our technique type.

The final part of this book covers applications of deep learning for computer vision *outside* of image classification, including basic object detection, deep dreaming and neural style, Generative Adversarial Networks (GANs), and Image Super Resolution. Again, the techniques covered in this volume are meant to be *much* more advanced than the *Starter Bundle* – this is where you'll start to separate yourself from a *deep learning novice* and transform into a true ***deep learning practitioner***. To start your transformation to deep learning expert, just flip the page.

2. Introduction

Welcome to the *ImageNet Bundle* of *Deep Learning for Computer Vision with Python*, the final volume in the series. This volume is meant to be the most *advanced* in terms of content, covering techniques that will enable you to reproduce results of state-of-the-art publications, papers, and talks. To help keep this work organized, I've structured the *ImageNet Bundle* in two parts.

In the first part, we'll explore the ImageNet dataset in detail and learn how to train state-of-the-art deep networks including AlexNet, VGGNet, GoogLeNet, ResNet, and SqueezeNet *from scratch*, obtaining as similar accuracies as possible as their respective original works. In order to accomplish this goal, we'll need to call on all of our skills from the *Starter Bundle* and *Practitioner Bundle*.

We'll need to ensure we understand the fundamentals of Convolutional Neural Networks, especially layer types and regularization, as we implement some of these more "exotic" architectures. Luckily, you have already seen more shallow implementations of these deeper architectures inside the *Practitioner Bundle* so implementing networks such as VGGNet, GoogLeNet, and ResNet will feel somewhat familiar.

We'll also need to ensure we are comfortable with *babysitting the training process* as we can easily overfit our network architectures on the ImageNet dataset, *especially* during the later epochs. Learning how to correctly monitor loss and accuracy plots to determine if/when parameter updates should be updated is an acquired skill, so to help you develop this skill faster and train deep architectures on large, challenging datasets, I've written each of these chapters as "experiment journals" that apply the scientific method.

Inside each chapter for a given network you'll find:

1. The exact process I used when training the network.
2. The particular results.
3. The changes I decided to make in the next experiment.

Thus, each chapter reads like a "story": you'll find out what worked for me, what didn't, and ultimately what obtained the best results and enabled me to replicate the work of a given publication. After reading this book, you'll be able to use this knowledge to train your own network architectures from scratch on ImageNet *without* spinning your wheels and wasting weeks (or even months) of time trying to tune your parameters.

The second part of this book focuses on case studies – real-world applications of applying deep learning and computer vision to solve a particular problem. We'll first start off by training a CNN from scratch to recognition emotions/facial expressions of people in real-time video streams. From there we'll use transfer learning via feature extraction to automatically detect and correct image orientation. A second case study on transfer learning (this time via fine-tuning) will enable us to recognize over 164 vehicle makes and models in images. A model such as this one could enable you to create an "intelligent" highway billboard system that displays targeted information or advertising to the driver based on what type of vehicle they are driving. Our final case study will demonstrate how to train a CNN to correctly predict the age and gender of a person in a photo.

Finally, I want to remind you that the techniques covered in this volume are *much* more advanced than both the *Starter Bundle* and the *Practitioner Bundle*. Both of the previous volumes gave you the required knowledge you need to be successful when reading through this book – but this point is where you'll separate yourself from a *deep learning practitioner* and a true *deep learning master*. To start your final transformation into a deep learning expert, just flip the page.

3. Training Networks Using Multiple GPUs

Training deep neural networks on large scale datasets can take a long time, even single experiments can take *days* to finish. In order to speed up the training process, we can use *multiple GPUs*. While backends such as Theano and TensorFlow (and therefore Keras) do support multiple GPU training, the process to set up a multiple GPU experiment is arduous and non-trivial. I do expect this process to change for the better in the future and become substantially easier.

Therefore, for deep neural networks and large datasets, I *highly recommend* using the mxnet library [5] which we will be using for the majority of experiments in the remainder of this book. The mxnet deep learning library (written in C++) provides bindings to the Python programming language and specializes in *distributed, multi-machine learning* – the ability to parallelize training across GPUs/devices/nodes is critical when training state-of-the-art deep neural network architectures on massive datasets (such as ImageNet).

The mxnet library is also very easy to work with – given your background using the Keras library from previous chapters in this book, you'll find working with mxnet to be easy, straightforward, and even quite natural.

It's important to note that all neural networks in the *ImageNet Bundle* can be trained using a single GPU – *the only caveat is time*. Some networks, such as AlexNet and SqueezeNet, require only a few days time to be trained on a single GPU. Other architectures, such as VGGNet and ResNet, may take over a month to train on a single GPU.

In the first part of this chapter, I'll highlight the network architectures we'll be discussing that can easily be trained on a single GPU and which architectures should use multiple GPUs if at all possible. Then, in the second half of this chapter, we'll examine some of the performance gains we can expect when training Convolutional Neural Networks using multiple GPUs.

3.1 How Many GPUs Do I Need?

If you were to ask any seasoned deep learning practitioner how many GPUs you need to train a reasonably deep neural network on a large dataset, their answer would almost always be *"The more, the better"*. The benefit of using multiple GPUs is obvious – *parallelization*. The more GPUs we

can throw at the problem, the faster we can train a given network. However, some of us may only have *one* GPU when working through this book. That raises the questions:

- Is using just *one* GPU a fruitless exercise?
- Is reading through this chapter a waste a time?
- Was purchasing the *ImageNet Bundle* a poor investment?

The answer to all of these questions is a resounding *no* – you are in good hands, and the *knowledge you learn here will be applicable to your own deep learning projects*. However, you do need to *manage your expectations* and realize you are crossing a threshold, one that separates *educational* deep learning problems from *advanced, real-world applications*. **You are now entering the world of state-of-the-art deep learning** where experiments can take days, weeks, or even in some rare cases, months to complete – *this timeline is totally and completely normal*.

Regardless if you have one GPU or eight GPUs, you'll be able to replicate the performance of the networks detailed in this chapter, but again, keep in mind the caveat of time. The more GPUs you have, the faster the training will be. If you have a single GPU, don't be frustrated – simply be patient and understand this is part of the process. The primary goal of the *ImageNet Bundle* is to provide you with actual case studies and detailed information on how to train state-of-the-art deep neural networks on the challenging ImageNet dataset (along with a few additional applications as well). No matter if you have one GPU or eight GPUs, *you'll be able to learn from these case studies* and use this knowledge in your own applications.

For readers using a single GPU, I highly recommend spending most of your time training AlexNet and SqueezeNet on the ImageNet dataset. These networks are more shallow and can be trained much faster on a single GPU system (in the order of 3-6 days for AlexNet and 7-10 days for SqueezeNet, depending on your machine). Deeper Convolutional Neural Networks such as GoogLeNet can also be trained on a single GPU but can take up to 7-14 days.

Smaller variations of ResNet can also be trained on a single GPU as well, but for the deeper version covered in this book, I would recommend multiple GPUs.

The *only* network architecture I *do not* recommend attempting to train using one GPU is VGGNet – not only can it be a pain to tune the network hyperparameters (as we'll see later in this book), but the network is *extremely slow* due to its depth and number of fully-connected nodes. If you decide to train VGGNet from scratch, keep in mind that it can take up to 14 days to train the network, even using four GPUs.

Again, as I mentioned earlier in this section, *you are now crossing the threshold* from deep learning practitioner to deep learning expert. The datasets we are examining are *large* and *challenging* – and the networks we will train on these datasets are *deep*. As depth increases, so does the computation required to perform the forward and backward pass. Take a second now to set your expectations that these experiments are *not* ones you can leave running overnight and gather the results the next morning – your experiments *will* take longer to run. **This is a fact that every deep learning researcher must accept.**

But even if you are training your own state-of-the-art deep learning models on a single GPU, don't fret. The same techniques we use for multiple GPUs can also be applied to single GPUs. The sole purpose of the *ImageNet Bundle* is to give you the *knowledge* and *experience* you need to be successful applying deep learning to your own projects.

3.2 Performance Gains Using Multiple GPUs

In an ideal world, if a single epoch for a given dataset and network architecture takes N seconds to complete on a single GPU, then we would expect the same epoch with two GPUs to complete in $N/2$ seconds. However, this expectation isn't the actual case. Training performance is *heavily* dependent on the PCIe bus on your system, the specific architecture you are training, the number of layers in the network, and whether your network is bound via *computation* or *communication*.

In general, training with two GPUs tends to improve speed by $\approx 1.8x$. When using four GPUs, performance scales to $\approx 2.5 - 3.5x$ scaling depending on your system [6]. Thus, training does not decrease linearly with the number of GPUs on your system. Architectures that are bound by *computation* (larger batch sizes increasing with the number of GPUs) will scale better with multiple GPUs as opposed to networks that rely on *communication* (i.e., smaller batch sizes) where latency starts to play a role in degrading performance.

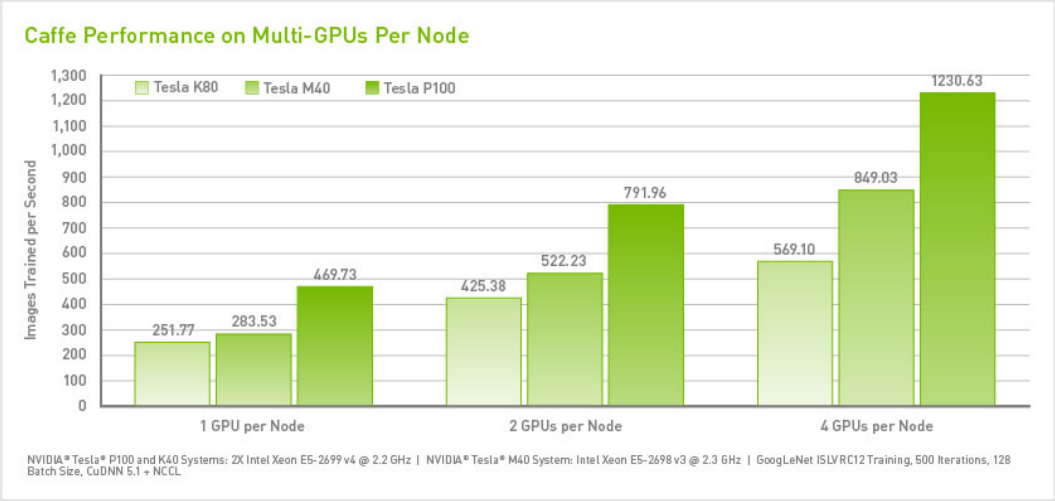


Figure 3.1: On the x-axis we have the number of GPUs (1-4) while the y-axis describes the number of images trained per second. When increasing from one to two GPUs we can inspect an increase of $\approx 1.82x$ performance. Jumping from one to four GPUs yields an increase of ≈ 2.71.

To investigate GPU scaling further, let's look at the official benchmarks released by NVIDIA in Figure 3.1. Here we can see three types of GPUs (Tesla K80, Tesla M40, and Tesla P400) that are used to train GoogLeNet on the ImageNet dataset using the Caffe [7] deep learning library. The x-axis plots the number of GPUs (one, two, and four, respectively) while the y-axis describes the number of images trained per second (forward and backward pass). On average, we see a performance increase of $\approx 1.82x$ when switching from one GPU to two GPUs. When comparing one GPU to four GPUs, performance increases to $\approx 2.71x$.

Performance will continue to increase as more GPUs are added to the system, but again, keep in mind that training speed will not scale linearly with the number GPUs – if you train a network using one GPU, then train it again using four GPUs, don't expect the amount of time it takes to train the network to decrease by a factor of four. That said, there are performance gains to be had by training deep learning models with more GPUs, so if you have them available, by all means, *use them*.

3.3 Summary

In this chapter, we discussed the concept of training deep learning architectures using *multiple* GPUs. To perform most of the experiments in this book, we'll be using the mxnet library which is highly optimized for multi-GPU training. Given your experience using the Keras library throughout earlier chapters in this book, you'll find using mxnet to be natural with function and class names being very similar.

From there we discussed basic expectations when training networks using a single GPU versus multiple GPUs. Yes, training a deep network on a large dataset with a single GPU *will* take longer,

but don't be discouraged – the same techniques you use for single GPU instances will apply to multi-GPU instances as well. Keep in mind that you are now crossing the threshold from *deep learning practitioner* to *deep learning expert* – the experiments we perform here *will* be more challenging and will require more time and effort. Set this expectation now as all deep learning researchers do in their career.

In Chapter 6, we'll train our first Convolutional Neural Network, AlexNet, on the ImageNet dataset, replicating the performance of Krizhevsky et al. in their seminal work in 2012 [8], which changed the landscape of image classification forever.

4. What Is ImageNet?

In this chapter, we'll discuss the ImageNet dataset and the associated ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [9]. This challenge is the *de facto* benchmark for evaluating image classification algorithms. The leaderboard for ILSVRC has been *dominated* by Convolutional Neural Networks and deep learning techniques since 2012 when Krizhevsky et al. published their seminal AlexNet work [8].

Since then, deep learning methods have continued to widen the accuracy gap between CNNs and other traditional computer vision classification methods. There is no doubt that CNNs are powerful image classifiers and are now a permanent fixture in the computer vision and machine learning literature. In the second half of this chapter, we'll explore how to obtain the ImageNet dataset, a requirement in order for you to replicate the results of state-of-the-art neural networks later in this chapter.

4.1 The ImageNet Dataset

Within the computer vision and deep learning communities, you might run into a bit of contextual confusion surrounding what ImageNet *is* and *isn't*. ImageNet is actually a *project* aimed at labeling and categorizing images into all its 22,000 categories based on a defined set of words and phrases. At the time of this writing, there are over *14 million* images in the ImageNet project.

So, how is ImageNet organized? To order such a massive amount of data, ImageNet actually follows the WordNet hierarchy [10]. Each meaningful word/phrase inside WordNet is called a "synonym set" or *synset* for short. Within the ImageNet project, images are categorized according to these synsets; the goal of the project is to have 1,000+ images per synset.

4.1.1 ILSVRC

In the context of computer vision and deep learning, whenever you hear people talking about image net, they are likely referring to the ImageNet Large Scale Visual Recognition Challenge [9], or simply ILSVRC for short. The goal of the image classification track in this challenge is to train a model that can correctly classify an image into *1,000 separate object categories*, some of which are considered fine-grained classification and others which are not. Images inside the ImageNet dataset

were gathered by compiling previous datasets and scraping popular online websites. These images were then manually labeled, annotated, and tagged.

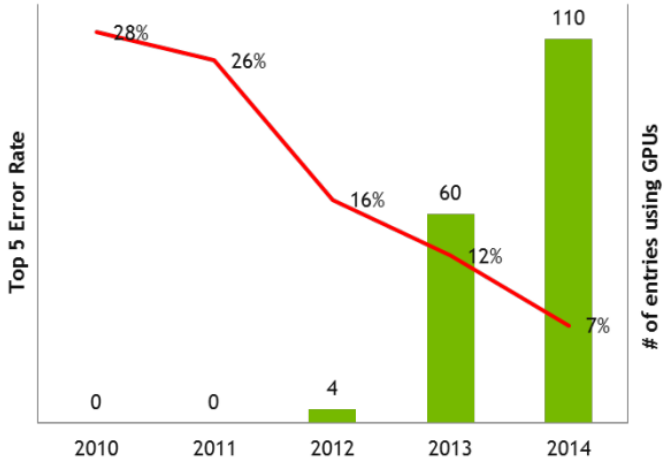


Figure 4.1: Since the seminal AlexNet architecture was introduced in 2012, Convolutional Neural Network methods have dominated the ILSVRC challenge, both in terms of accuracy and number of entries. (Image credit: NVIDIA [11])

Since 2012, the leaderboard for the ILSVRC challenges has been dominated by deep learning-based approaches with the rank-1 and rank-5 accuracies increasing every year (Figure 4.1). Models are trained on $\approx$ 1.2 million training images with another 50,000 images for validation (50 images per synset) and 100,000 images for testing (100 images per synset).

These 1,000 image categories represent various object classes that we may encounter in our day-to-day lives, such as species of dogs, cats, various household objects, vehicle types, and much more. You can find the full list of object categories in the ILSVRC challenge on this official ImageNet documentation page (http://pyimg.co/1ogm0).

In Chapter 5 of the *Starter Bundle*, I included a figure demonstrating some of the challenges associated with the ImageNet dataset from the *ImageNet Large Scale Visual Recognition Challenge*. Unlike having generic "bird", "cat", and "dog" classes, ImageNet includes more fine-grained classes compared to previous image classification benchmark datasets such as PASCAL VOC [12]. While PASCAL VOC limited "dog" to only a *single category*, ImageNet instead includes *120 different breeds* of dogs. This finger classification requirement implies that our deep learning networks not only need to recognize images as "dog", but also be *discriminative enough* to determine *what species of dog*.

Furthermore, images in ImageNet vary *dramatically* across object scale, number of instances, image clutter/occlusion, deformability, texture, color, shape, and real-world size. This dataset is *challenging*, to say the least, and in some cases, it's hard for even humans to correctly label. Because of the challenging nature of this dataset, deep learning models that perform well on ImageNet are likely to generalize well to images outside of the validation and testing set – this is the exact reason why we apply transfer learning to these models as well.

We'll discuss more examples of images and specific classes in Chapter 5 when we start exploring the ImageNet dataset and write code to prepare our images for training. However, until that time, I would highly encourage you to take 10-20 minutes and browse the synsets (http://pyimg.co/1ogm0) in your web browser to get a feel for the *scale* and *challenge* associated with correctly classifying these images.

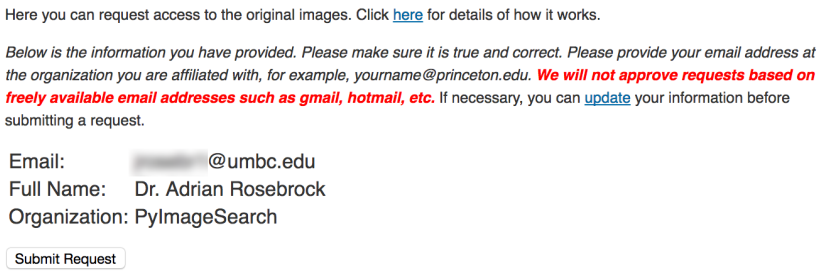
4.2 Obtaining ImageNet

The ImageNet classification challenge dataset is quite large, weighing in at 138GB for the training images, 6.3GB for the validation images, and 13GB for the testing images. Before you can download ImageNet, you first need to obtain access to the ILSVRC challenge and download the images and associated class labels. This section will help you obtain the ImageNet dataset.

4.2.1 Requesting Access to the ILSVRC Challenge

The ILSVRC challenge is a joint work between Princeton and Stanford universities, and is, therefore, an academic project. ImageNet does not own the copyrights to the images and only grants access to the *raw image files* for non-commercial research and/or educational purposes (although this point is up for debate – see Section 4.2.5 below). If you fall into this camp, you can simply register for an account on the ILSVRC website (http://pyimg.co/fy844).

However, please note that ImageNet does not accept freely available email addresses such as Gmail, Yahoo, etc. – instead, you will need to supply the email address of your university or government/research affiliation. As Figure 4.2 demonstrates, I simply needed to provide my university email address, from there I was able to verify my email address and then accept the Terms of Access.


Here you can request access to the original images. Click here for details of how it works.

Below is the information you have provided. Please make sure it is true and correct. Please provide your email address at the organization you are affiliated with, for example, yourname@princeton.edu. ***We will not approve requests based on freely available email addresses such as gmail, hotmail, etc.*** If necessary, you can update your information before submitting a request.

Email: [redacted]@umbc.edu
Full Name: Dr. Adrian Rosebrock
Organization: PyImageSearch

Submit Request

Figure 4.2: If you have a university or research organization associated email address, be sure to use it when registering with ImageNet and the associated ILSVRC competition.

Once you've accepted the Terms of Access you'll have access to the *Download Original Images* page – click the ILSVRC 2015 image data link. From there make sure you download the *Development Kit*, a .zip file containing a README, information on the training/testing splits, blacklisted files that should not be used for training, etc. (Figure 4.3).

You'll then want to download the CLS-LOC dataset which contains the 1.2 million images in the ImageNet dataset (Figure 4.3). Keep in mind that this is a *large* file and depending on your internet connection (and the stability of image-net.org), this download may take a couple of days. My personal suggestion would be to use the `wget` command line program to download the archive, enabling you to restart the download from where you left off, just in case there are connectivity issues (which there are likely to be a handful of). Explaining how to use `wget` is outside the scope of this book, so please refer to the following page for instructions on how to restart a download with `wget` (http://pyimg.co/97u59).

After the .tar archive is downloaded, the next step is to unpack it, which is also a computationally expensive process as you need to unarchive $\approx$ 1.2 million images – I would suggest leaving your system to tackle this task overnight.

4.2.2 Downloading Images Programmatically

If you are denied access to the ILSVRC raw image data, don't worry – there are other methods to obtain the data, although those methods are slightly more tedious. Keep in mind that ImageNet

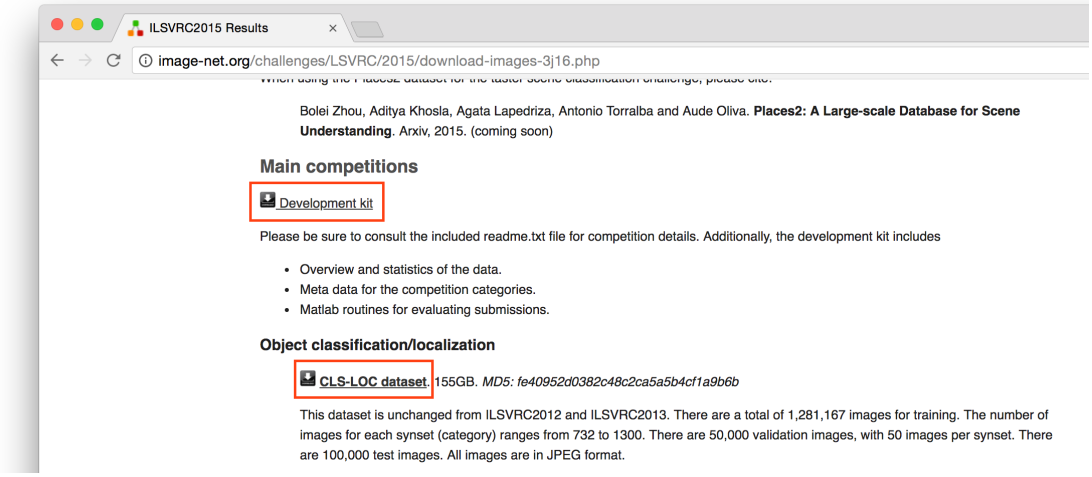


Figure 4.3: To download the entire ImageNet dataset you'll need to download the development kit along with the large .tar archive of the $\approx 1.2 million$ images. (highlighted with red rectangles).

does not "own" the images inside the dataset so they can freely distribute the URLs of the images. The URLs of every image (a .txt file with one URL per line) in the dataset can be found here:

http://pyimg.co/kw64x

Again, you would need to use `wget` to download the images. A common problem you may encounter here is that some image URLs may have naturally 404'd since the original web crawl and you won't have access to them. Therefore, downloading the images programmatically can be quite cumbersome, tedious, and a method I do not recommend. But don't fret – there is another way to obtain ImageNet.

4.2.3 Using External Services

Due to the vast size of the ImageNet dataset and the need for it to be distributed globally, the dataset lends itself well to being distributed via BitTorrent. The website AcademicTorrents.com provides downloads for both the training set and validation set (http://pyimg.co/asdyi) [13]. A screenshot of the webpage can be found in Figure 4.4.

The testing set is *not* included in the torrent as we will not have access to the ImageNet evaluation server to submit our predictions on the testing data. Please keep in mind that even if you *do* use external services such as AcademicTorrents to download the ImageNet dataset, you are *still* implicitly bound to the Terms of Access. You can use ImageNet for researching and developing your own models, but you *cannot* repackage ImageNet and use it for profit – this is *strictly* an academic dataset provided by a joint venture between Stanford and Princeton. Respect the scientific community and do not violate the Terms of Access.

4.2.4 ImageNet Development Kit

While you are downloading the actual ImageNet dataset, make sure you download the ImageNet Development Kit (http://pyimg.co/wijj7) which we'll henceforth simply refer to as "DevKit".

I have also placed a mirror to the DevKit here: http://pyimg.co/ounw6

The DevKit contains:

- An overview and statistics for the dataset.
- Meta data for the categories (allowing us to build our image filename to class label mappings).
- MATLAB routines for evaluation (which we will not need).

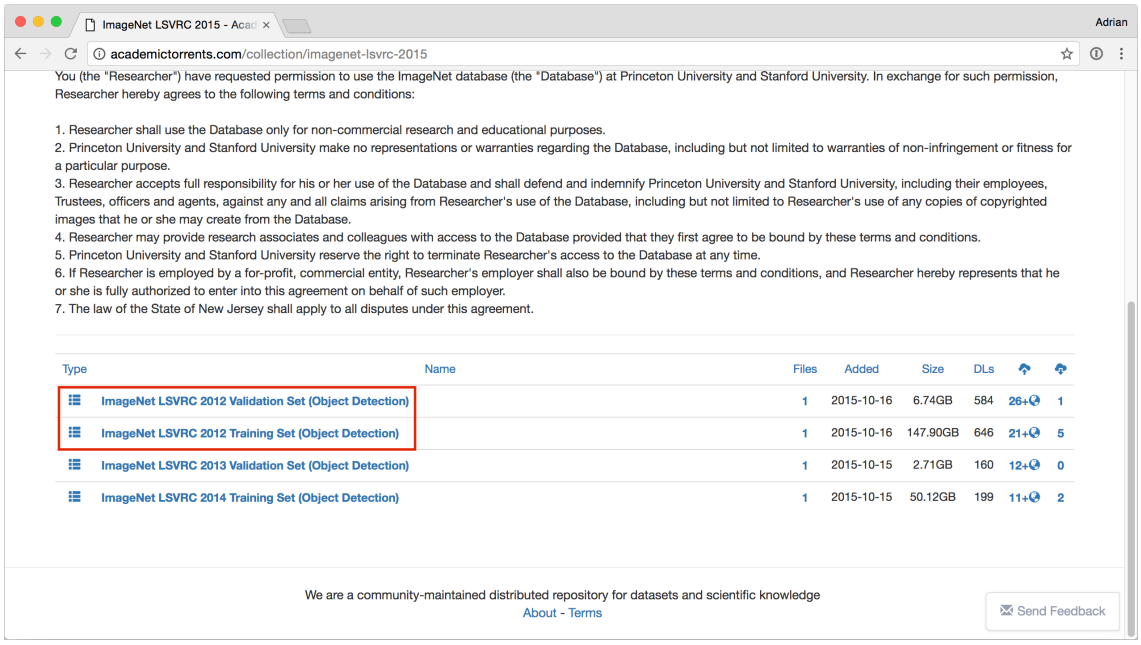


Figure 4.4: A screenshot of the AcademicTorrents website for ImageNet. Make sure you download the *"ImageNet LSVRC 2012 Validation Set (Object Detection)"* and *"ImageNet LSVRC 2012 Training Set (Object Detection)"* files, outlined in red.

The DevKit is a small download at only 7.4MB and should complete within a few seconds. Once you've downloaded the DevKit, unarchive it, and take the time to familiarize yourself with the directory structuring including the license (`COPYING`) and the `readme.txt`. We'll be reviewing the DevKit in detail in our following chapter when we build the ImageNet dataset and prepare it for training a CNN.

4.2.5 ImageNet Copyright Concerns

At first glance, it may appear that the ImageNet dataset and associated ILSVRC challenge is a minefield of copyright claims – *who* exactly owns *what* in the ImageNet dataset? To answer this question, let's break the problem into three concrete asset classes:

- **Asset #1:** The images themselves.
- **Asset #2:** The pre-compiled ILSVRC dataset.
- **Asset #3:** The output model weights obtained by training a network on ILSVRC.

To start, the raw images themselves belong to the person/entity who captured the image – *they own the full copyright over these images*. The ImageNet project operates under the same restrictions as search engines like Google, Bing, etc. – they are allowed to provide *links* to the original copyrighted images, *provided* that the copyright is retained. This provision is why the ImageNet website is allowed to provide the URLs of the original images in the dataset *without* requiring you to register and create an account – it is your responsibility to actually download them.

This process seems fairly clear-cut; however, the waters start to muddy once we look at the actual ILSVRC challenge. Since the end user is no longer responsible for downloading each image one-by-one (and can instead download an entire archive of the dataset), we run into copyright concerns – why can a user download a *pre-compiled archive* of (potentially) copyrighted images? Doesn't that violate the copyright of the person who took the original photograph? This is a point of debate between the arts and sciences communities, but as it currently stands, we are allowed to

download the ILSVRC image archives due to the Terms of Access we accept when participating in ILSVRC:

1. You are free to use the ImageNet dataset for academic and non-commercial pursuits.
2. You *cannot* distribute the ILSVRC data as part of your end product.

The original copyright question is not answered directly but is somewhat pacified by the restrictions placed on the pre-compiled dataset archives. Furthermore, the ImageNet website provides DMCA takedown applications for copyright holders who wish to have their images removed from the dataset.

Finally, let's examine asset #3, a given model's serialized weights obtained after training a Convolutional Neural Network on the ImageNet dataset – *are these model weights under copyright as well?*

The answer is a bit unclear, but as far as our current understanding of the law goes, there is no restriction on the open release of learned model weights [14] Therefore, we are free to distribute our trained models as we see fit, provided we keep in mind the spirit of fair use and proper attribution.

The reason we are allowed to distribute our own models (and even copyright them using our own restrictions) is due to *parameterized learning* (*Starter Bundle*, Chapter 8) – our CNN does not store "internal copies" of the raw images (such as the k-NN algorithm would). Since the model does not store the original images (whether in whole or part), the model *itself* is not bound to the same copyright claims as the original ImageNet dataset. We can thus distribute our model weights freely or place additional copyrights on them (for example, the end user is free to use our existing architecture, but must re-train the network from scratch on the original dataset *before* using it in a commercial application).

But what about models *trained on ImageNet* that are used in *commercial applications?*

Do models trained on the ImageNet dataset *and* used in commercial applications violate the Terms of Access? According to the Terms of Access wording, yes, technically these commercial applications are at risk of breaking the contract.

On the other hand, there has been no lawsuit brought against a deep learning company/startup who has trained their own networks from scratch using the ImageNet dataset. Keep in mind that a copyright has no power *unless it's enforced* – no such enforcing has ever been done regarding ImageNet.

In short: **this is a gray area in the deep learning community**. There are a large number of deep learnings startups that rely on CNNs trained on the ImageNet dataset (company names omitted on purpose) – their revenue is based solely on the performance of these networks. In fact, without ImageNet and ILSVRC, these companies wouldn't have the dataset required to create their product (unless they invested millions of dollars and many years collecting and annotating the dataset themselves).

My Anecdotal Opinion

It is my ***anecdotal opinion*** that there is an `unspoken set of rules` that govern the fair usage of the ImageNet dataset. **I believe these rules to be as follows** (although there are sure to be many who disagree with me):

- **Rule #1:** You need to obtain the ILSVRC dataset by some means and accept (either explicitly or implicitly) the Terms of Access.
- **Rule #2:** After obtaining the data associated with the ILSVRC challenge, you need to train your own Convolutional Neural Network on the dataset. You are free to use *existing* network architectures such as AlexNet, VGGNet, ResNet, etc. *provided* that you train the network from scratch on the ILSVRC dataset. You *do not* need to develop a novel network architecture.
- **Rule #3:** Once you have obtained your model weights, you can then distribute them under your own restrictions, including open access, usage with attribution, and even limited com-

mercial usage.

Rule number three will be hotly contested, and I'm positive I'll receive a number of emails about it – but the point is this – while the rules are unclear, there have been *no lawsuits* brought to court on how network weights *derived* from ILSVRC can be used, *including* commercial applications. Again, keep in mind that a copyright is only valid if it is actually enforced – simply *holding* a copyright does not serve as a form of protection.

Furthermore, the usage of deep learning models trained on ILSVRC is both a legal issue and an economic issue – the computer science industry is experiencing a *tremendous boom* in deep learning applications. If sweeping legislation were to be passed restricting the commercial usage of CNNs trained from scratch on copyrighted image data (even though there are no replicates of the original data due to parameterized learning), we would be killing part of an economy experiencing *high growth* and *valuations in the billions of dollars*. This style of highly restrictive, sweeping legislature could very easily catalyze another AI winter (*Starter Bundle*, Chapter 2).

For further information on "who owns what" in the deep learning community (datasets, model weights, etc.), take a look *Deep Learning vs. Big Data: Who owns what?*, an excellent article by Tomasz Malisiewicz on the subject [15].

4.3 Summary

In this chapter, we reviewed the ImageNet dataset and associated ILSVRC challenge, the *de facto* benchmark used to evaluate image classification algorithms. We then examined multiple methods to obtain the ImageNet dataset.

In the remaining chapters in this book I will assume that you *do not* have access to the testing set and associated ImageNet evaluation server; therefore, we will derive our own testing set from the training data. Doing so will ensure we can evaluate our models locally and obtain a reasonable proxy to the accuracy of our network.

Take the time now to start downloading the ImageNet dataset on your machine. I would recommend using the official ILSVRC challenge website to download the ImageNet data as this method is the easiest and most reliable. If you do not have access to a university, government, or research affiliated email address, feel free to ask your colleagues for access – but again keep in mind that you are still bound to the Terms of Access, regardless of how you obtain the data (even if you download via AcademicTorrents).

It is my *anecdotal opinion* that models weights obtained via training on the ILSVRC dataset can be used as you best see fit; however, keep in mind that this is still a point of contention. Before deploying a commercial application that leverages a model trained on ImageNet, I would encourage you to consult proper legal counsel.

In our next chapter, we'll explore the ImageNet dataset, understand its file structure, and write Python helper utilities to facilitate our ability to load the images from disk and prepare them for training.

5. Preparing the ImageNet Dataset

Once you've downloaded the ImageNet dataset, you might be a bit overwhelmed. You now have over 1.2 million images residing on disk, none of them have "human readable" file names, there isn't an obvious way to extract the class labels from them, and it's totally unclear how you are supposed to train a custom Convolutional Neural Network on these images – *what have you gotten yourself into?*

No worries, I've got you covered. In this chapter, we'll start by understanding the ImageNet file structure, including both the raw images along with the development kit (i.e., "DevKit"). From there, we'll write a helper Python utility script that will enable us to parse the ImageNet filenames + class labels, creating a nice output file that maps a given input filename to its corresponding label (one filename and label per line).

Finally, we'll use these output files along with the mxnet `im2rec` tool, which will take our mappings and create *efficiently packed* record (`.rec`) files that can be used when training deep learning models on datasets too large to fit into main memory. As we'll find out, this `.rec` format is not only more *compact* than HDF5, but it's also more I/O efficient as well, enabling us to train our networks faster.

The techniques and tools we apply in this chapter will allow us to train our own custom CNNs from scratch on the ImageNet dataset in subsequent chapters. In later chapters, such as our case studies on vehicle make and model identification along with age and gender prediction, we'll again use these same tools to help us create our image datasets.

Be sure to pay close attention to this dataset and take your time when working through it. The code we will be writing isn't necessarily "deep learning code", but rather helpful utility scripts that will *facilitate* our ability to train networks further down the road.

5.1 Understanding the ImageNet File Structure

Let's go ahead and get started by understanding the ImageNet file structure. I'll assume that you have finished downloading the `ILSVRC2015_CLS-LOC.tar.gz` file, likely having to restart the download at least two to three times (when I personally downloaded the massive 166GB archive, it

took two restarts and a total of 27.15 hours to download). I then unpacked the archive using the following command:

```
$ tar -xvf ILSVRC2015_CLS-LOC.tar.gz
```

I would suggest starting this command right before you go to bed to ensure it has been fully unpacked by the time you wake up in the morning. Keep in mind that there are over 1.2 million images in the training set alone, so the unarchive process will take a bit of time.

Once the tarball has finished uncompressing, you'll have a directory named `ILSVRC2015`:

```
$ ls ILSVRC2015
```

Let's go ahead and change directory into `ILSVRC2015` and list the contents, where you'll find three sub-directories:

```
$ cd ILSVRC2015
$ ls
Annotations  Data  ImageSets
```

First, we have the `Annotations` directory. This directory is only used for the *localization challenge* (i.e., object detection), so we can ignore this directory.

The `Data` directory is more important. Inside `Data` we'll find a sub-directory named `CLS-LOC`:

```
$ ls Data/
CLS-LOC
```

Here we can find the training, testing, and validation "splits":

```
$ ls Data/CLS-LOC/
test  train  val
```

I put the word "splits" in quotations as there is still work that needs to be done in order to get this data in a format such that we can train a Convolutional Neural Network on it and obtain state-of-the-art classification results. Let's go ahead and review each of these sub-directories individually.

5.1.1 ImageNet "test" Directory

The `test` directory contains (as the name applies) 100,000 images (100 data points for each of the 1,000 classes) for our testing split:

```
$ ls -l Data/CLS-LOC/test/ | head -n 10
total 13490508
-rw-r--r-- 1 adrian adrian   33889 Jul  1  2012 ILSVRC2012_test_00000001.JPEG
-rw-r--r-- 1 adrian adrian  122117 Jul  1  2012 ILSVRC2012_test_00000002.JPEG
-rw-r--r-- 1 adrian adrian   26831 Jul  1  2012 ILSVRC2012_test_00000003.JPEG
-rw-r--r-- 1 adrian adrian  124722 Jul  1  2012 ILSVRC2012_test_00000004.JPEG
-rw-r--r-- 1 adrian adrian   98627 Jul  1  2012 ILSVRC2012_test_00000005.JPEG
```

```
-rw-r--r-- 1 adrian adrian  211157 Jul  1  2012 ILSVRC2012_test_00000006.JPEG
-rw-r--r-- 1 adrian adrian  219906 Jul  1  2012 ILSVRC2012_test_00000007.JPEG
-rw-r--r-- 1 adrian adrian  181734 Jul  1  2012 ILSVRC2012_test_00000008.JPEG
-rw-r--r-- 1 adrian adrian   10696 Jul  1  2012 ILSVRC2012_test_00000009.JPEG
```

However, we were unable to use these images directly for our experiments. Recall that the ILSVRC challenge is the *de facto* standard for image classification algorithms. In order to keep this challenge fair (and to ensure no one cheats), the labels for the testing set are kept private.

First, a person/team/organization trains their algorithm using the training and testing splits. Once they are satisfied with the results, predictions are made on the testing set. The predictions from the testing set are then automatically uploaded to the ImageNet evaluation server where they are compared to the ground-truth labels. At *no point* do any of the competitors have access to the testing ground-truth labels. The ImageNet evaluation server then returns their overall accuracy.

Some readers of this book may have access to the ImageNet evaluation server, in which case I encourage you to explore this format further and consider submitting your own predictions. However, many other readers will have obtained ImageNet without *directly* registering for an account on the ImageNet website. Either way is perfectly okay (provided you follow the licensing agreements I mentioned in Chapter 4), but you will not have access to the evaluation server. Since I want to keep this chapter as open and accessible to everyone, regardless of how you obtained ImageNet, we will ignore the `test` directory and create our own testing set by sampling the training data, just as we did for the Tiny ImageNet challenges in Chapter 11 and Chapter 12 of the *Practitioner Bundle*.

5.1.2 ImageNet "train" Directory

The `train` directory of ImageNet consists of a set of sub-directories:

```
$ ls -l Data/CLS-LOC/train/ | head -n 10
total 60020
drwxr-xr-x 2 adrian adrian 69632 Sep 29  2014 n01440764
drwxr-xr-x 2 adrian adrian 69632 Sep 29  2014 n01443537
drwxr-xr-x 2 adrian adrian 57344 Sep 29  2014 n01484850
drwxr-xr-x 2 adrian adrian 57344 Sep 29  2014 n01491361
drwxr-xr-x 2 adrian adrian 61440 Sep 29  2014 n01494475
drwxr-xr-x 2 adrian adrian 61440 Sep 29  2014 n01496331
drwxr-xr-x 2 adrian adrian 53248 Sep 29  2014 n01498041
drwxr-xr-x 2 adrian adrian 53248 Sep 29  2014 n01514668
drwxr-xr-x 2 adrian adrian 61440 Sep 29  2014 n01514859
```

At first, these sub-directory names may appear to be unreadable. However, recall from Chapter 4 on ImageNet that the dataset is organized according to WordNet IDs [10] called *synonym sets* or simply "syn sets" for short. A synset maps to a particular concept/object, such as *goldfish*, *bald eagle*, *airplane*, or *acoustic guitar*. Therefore, in each of these strangely labeled sub-directories, you will find approximately 732-1,300 images per class.

For example, the WordNet ID `n01440764` consists of 1,300 images of "tench", a type of European freshwater fish, closely related to the minnow family (Figure 5.1):

```
$ ls -l Data/CLS-LOC/train/n01440764/*.JPEG | wc -l
1300
$ ls -l Data/CLS-LOC/train/n01440764/*.JPEG | head -n 5
adrian   13697 Jun 10  2012 Data/CLS-LOC/train/n01440764/n01440764_10026.JPEG
```

```
adrian    9673 Jun 10  2012 Data/CLS-LOC/train/n01440764/n01440764_10027.JPEG
adrian   67029 Jun 10  2012 Data/CLS-LOC/train/n01440764/n01440764_10029.JPEG
adrian  146489 Jun 10  2012 Data/CLS-LOC/train/n01440764/n01440764_10040.JPEG
adrian    6350 Jun 10  2012 Data/CLS-LOC/train/n01440764/n01440764_10042.JPEG
```

Figure 5.1: A sample of 25 images from the `n01440764` syn-set. These images are examples of "tench", a type of European freshwater fish.

Given that the WordNet IDs of the training images are *built-into* the file name, along with the `train_cls.txt` file we are going to review later in this chapter, it will be fairly straightforward for us to associate a given training image with its class label.

5.1.3 ImageNet "val" Directory

Similar to the `test` directory, the `val` directory contains 50,000 images per class (50 images for each of the 1,000 classes):

```
$ ls -l Data/CLS-LOC/val/*.JPEG | wc -l
50000
```

Each of these 50,000 images are stored in a "flat" directory, implying that no extra sub-directories are used to help us associate a given image with a class label:

Furthermore, by inspecting the filenames, you can see that there are no class label identifying information built into the file paths (such as WordNet ID, etc.). Luckily, we'll be reviewing a file named `val.txt` later in this chapter which provides us with the mappings from filename to class label.

```
$ ls -l Data/CLS-LOC/val/ | head -n 10
total 6648996
```

```
-rw-r--r-- 1 adrian adrian  109527 Jun 12  2012 ILSVRC2012_val_00000001.JPEG
-rw-r--r-- 1 adrian adrian  140296 Jun 12  2012 ILSVRC2012_val_00000002.JPEG
-rw-r--r-- 1 adrian adrian  122660 Jun 12  2012 ILSVRC2012_val_00000003.JPEG
-rw-r--r-- 1 adrian adrian   84885 Jun 12  2012 ILSVRC2012_val_00000004.JPEG
-rw-r--r-- 1 adrian adrian  130340 Jun 12  2012 ILSVRC2012_val_00000005.JPEG
-rw-r--r-- 1 adrian adrian  151397 Jun 12  2012 ILSVRC2012_val_00000006.JPEG
-rw-r--r-- 1 adrian adrian  165863 Jun 12  2012 ILSVRC2012_val_00000007.JPEG
-rw-r--r-- 1 adrian adrian  107423 Jun 12  2012 ILSVRC2012_val_00000008.JPEG
-rw-r--r-- 1 adrian adrian  114708 Jun 12  2012 ILSVRC2012_val_00000009.JPEG
```

5.1.4 ImageNet "ImageSets" Directory

Now that we've gone through the `train`, `test`, and `val` sub-directories, let's go up a level back to the `Annotations` and `Data` folders. Here you'll see a directory named `ImageSets`. Let's change directory to `ImageSets` and investigate it:

```
$ ls
Annotations  Data  ImageSets
$ cd ImageSets/
$ ls
CLS-LOC
$ cd CLS-LOC
$ ls
test.txt  train_cls.txt  train_loc.txt  val.txt
```

We can ignore the `test.txt` file since we will be constructing our own testing split from the training data. However, we need to take a look at both `train_cls.txt` (where the "cls" stands for "classification") and `val.txt`. These files contain the base filenames for the training images (1,281,167) along with the validation images (50,000). You can verify this fact using the following command:

```
$ wc -l train_cls.txt val.txt
 1281167 train_cls.txt
   50000 val.txt
 1331167 total
```

This file reports a total of 1,331,167 images to work with. Investigating `train_cls.txt`, you can see the contents are simply a base image filename (without the file extension) and a unique integer ID with one row per line:

```
$ head -n 10 train_cls.txt
n01440764/n01440764_10026 1
n01440764/n01440764_10027 2
n01440764/n01440764_10029 3
n01440764/n01440764_10040 4
n01440764/n01440764_10042 5
n01440764/n01440764_10043 6
n01440764/n01440764_10048 7
n01440764/n01440764_10066 8
n01440764/n01440764_10074 9
n01440764/n01440764_10095 10
```

The base image filename will allow us to derive the *full* image filename. The unique integer is simply a counter that increments, one per row.

The same is true for `val.txt` as well:

```
$ head -n 10 val.txt
ILSVRC2012_val_00000001 1
ILSVRC2012_val_00000002 2
ILSVRC2012_val_00000003 3
ILSVRC2012_val_00000004 4
ILSVRC2012_val_00000005 5
ILSVRC2012_val_00000006 6
ILSVRC2012_val_00000007 7
ILSVRC2012_val_00000008 8
ILSVRC2012_val_00000009 9
ILSVRC2012_val_00000010 10
```

The unique integer ID for the image isn't too helpful, other than when we need to determine "blacklisted" images. Images that are marked as "blacklisted" by the ImageNet dataset curators are too *ambiguous* in their class labels, and therefore should not be considered in the evaluation process. Later in this chapter we'll loop over all blacklisted images and remove them from our validation set by examining the unique integer ID associated with each validation image.

The benefit of using the `train_cls.txt` and `val.txt` files is that we *do not* have to list the contents of the training and validation subdirectories using `paths.list_images` – instead, we can simply loop over each of the rows in these `.txt` files. We'll be using both of these files later in this chapter, when we convert the raw ImageNet files to `.rec` format, suitable for training with mxnet.

5.1.5 ImageNet "DevKit" Directory

Besides downloading the raw images themselves in Chapter 4, you also downloaded the ILSVRC 2015 DevKit. This archive contains the actual index files we need to map image file names to their corresponding class labels. You can unarchive the `ILSVRC2015_devkit.tar.gz` file using the following command:

```
$ tar -xvf ILSVRC2015_devkit.tar.gz
```

Inside the `ILSVRC2015` directory you'll find the directory we are looking for – `devkit`:

```
$ cd ILSVRC2015
$ ls
devkit
```

You can *technically* place this file anywhere you like on your system (as we'll be creating a Python configuration file to point to important path locations); however, I personally like to keep it with the `Annotations`, `Data`, and `ImageSets` sub-directories for organizational purposes. I would suggest you copy the `devkit` directory so that it lives with our `Annotations`, `Data`, and `ImageSets` directories as well:

```
$ cp -R ~/home/ILSVRC2015/devkit /raid/datasets/imagenet/
```

The exact paths you specify here will be dependent on your system. I'm simply showing the example commands I ran on my *personal* system. The actual *commands* you use will be the same, but you will need to update the corresponding file paths.

Let's go ahead and take a look at the contents of `devkit`:

```
$ cd devkit/
$ ls
COPYING  data  evaluation  readme.txt
```

You can read the `COPYING` file for more information on copying and distributing the ImageNet dataset and associated evaluation software. The `readme.txt` file contains information on the ILSVRC challenge, including how the dataset is structured (we are providing a more detailed review of the dataset in this chapter). The `evaluation` directory, as the name suggests, contains MATLAB routines for evaluating predictions made on the testing set – since we will be deriving our own testing set, we can ignore this directory.

Most importantly, we have the `data` directory. Inside `data` you'll find a number of metafiles, both in MATLAB and plain text (`.txt`) format:

```
$ cd data/
$ ls -l
total 2956
ILSVRC2015_clsloc_validation_blacklist.txt
ILSVRC2015_clsloc_validation_ground_truth.mat
ILSVRC2015_clsloc_validation_ground_truth.txt
ILSVRC2015_det_validation_blacklist.txt
ILSVRC2015_det_validation_ground_truth.mat
ILSVRC2015_vid_validation_ground_truth.mat
map_clsloc.txt
map_det.txt
map_vid.txt
meta_clsloc.mat
meta_det.mat
meta_vid.mat
```

From this directory, we are most concerned with the following three files:

- `map_clsloc.txt`
- `ILSVRC2015_clsloc_validation_ground_truth.txt`
- `ILSVRC2015_clsloc_validation_blacklist.txt`

The `map_clsloc.txt` file maps our *WordNet IDs* to *human readable class labels* and is, therefore, the easiest method to convert a WordNet ID to a label a human can interpret. Listing the first few lines of this file, we can see the mappings themselves:

```
$ head -n 10 map_clsloc.txt
n02119789 1 kit_fox
n02100735 2 English_setter
n02110185 3 Siberian_husky
n02096294 4 Australian_terrier
n02102040 5 English_springer
n02066245 6 grey_whale
n02509815 7 lesser_panda
```

```
n02124075 8 Egyptian_cat
n02417914 9 ibex
n02123394 10 Persian_cat
```

Here we can see the WordNet ID `n02119789` maps to the `kit_fox` class label. The `n02096294` WordNet ID corresponds to the `Australian_terrier`, a species of dog. This mapping continues for all 1,000 classes in the ImageNet dataset. As detailed in Section 5.1.3 above, the images inside the `val` directory do not contain any class label information built into the filename; however, we do have the `val.txt` file inside the `ImageSets` directory. The `val.txt` file lists the (partial) image filenames for the validation set. There are exactly 50,000 entries (on per line) in the `val.txt` file. There are also 50,000 entries (one per line) inside `ILSVRC2015_clsloc_validation_ground_truth.txt`.

Let's take a look at these entries:

```
$ head -n 10 ILSVRC2015_clsloc_validation_ground_truth.txt
490
361
171
822
297
482
13
704
599
164
```

As we can see, there is a single integer listed on each line. Taking the first line of `val.txt` and the first line of `ILSVRC2015_clsloc_validation_ground_truth.txt` we end up with:

```
(ILSVRC2012_val_00000001, 490)
```

Figure 5.2: This image is of a snake, but what kind of snake? To find out we need to examine the ground-truth labels for the validation set inside ImageNet.

If we were to open up `ILSVRC2012_val_00000001.JPEG` we would see the image in Figure 5.2. Clearly, this is some sort of snake – but what type of snake? If we examine `map_clsloc.txt`, we see that class label ID with 490 is WordNet ID `n01751748`, which is a `sea_snake`:

```
$ grep ' 490 ' map_clsloc.txt
n01751748 490 sea_snake
```

Therefore, we need to use both `val.txt` and `ILSVRC2015_clsloc_validation_ground_truth.txt` to build our validation set.

Let's also examine the contents of `ILSVRC2015_clsloc_validation_blacklist.txt`:

```
$ head -n 10 ILSVRC2015_clsloc_validation_blacklist.txt
36
50
56
103
127
195
199
226
230
235
```

As I mentioned before, some validation files are considered too *ambiguous* in their class label. Therefore, the ILSVRC organizers marked these images as "blacklisted", implying that they should *not* be included in the validation set. When building our validation set, we need to check the validation image IDs to this blacklist set – if we find that a given image belongs in this set, we'll ignore it and exclude it from the validation set.

As you can see, there are *many* files required to build the ImageNet dataset. Not only do we need the raw images themselves, but we also need a number of `.txt` files used to construct the mappings from the original training and validation filename to the corresponding class label. This would be a tough, arduous process to perform by hand, so in the next section, I will show you my `ImageNetHelper` class that I *personally* use when building the ImageNet dataset.

5.2 Building the ImageNet Dataset

The overall goal of building the ImageNet dataset is so that we can train Convolutional Neural Networks *from scratch* on it. Therefore, we will review building the ImageNet dataset in *context* of preparing it for a CNN. To do so, we'll first define a configuration file that stores all relevant image paths, plaintext paths, and any other settings we wish to include.

From there, we'll define a Python class named `ImageNetHelper` which will enable us to quickly and easily build:

1. Our `.lst` files for the training, testing, and validation split. Each line in a `.lst` file contains the unique image ID, class label, and the *full path* to the input image. We'll then be able to use these `.lst` files in conjunction with the mxnet tool `im2rec` to convert our image files to an efficiently packed record file.
2. Our mean Red, Green, and Blue channel averages for the training set which we'll later use when performing mean normalization.

5.2.1 Your First ImageNet Configuration File

Whenever training a CNN on ImageNet, we'll create a project with the following directory structure:

```
--- mx_imagenet_alexnet
|    |--- config
```

```
|    |    |--- __init__.py
|    |    |--- imagnet_alexnet_config.py
|    |--- imagenet
|    |--- output/
|    |--- build_imagenet.py
|    |--- test_alexnet.py
|    |--- train_alexnet.py
```

As the directory and filenames suggests, this configuration file is for AlexNet. Inside the `config` directory we have placed two files:

1. `__init__.py`
2. `imagenet_alexnet_config.py`

The `__init__.py` file turns `config` into a Python package that is actually *importable* via the `import` statement into our own scripts – this file enables us to use Python syntax/libraries *within* the actual configuration, making the process of configuring a neural network for ImageNet *much* easier. The actual ImageNet configurations are then stored in `imagenet_alexnet_config.py`.

Instead of typing out the full path to the ImageNet dataset (i.e., `/raid/datasets/imagenet/`) I decided to create a symbolic link (often called "sym-link" or a "shorcut") aliased as `imagenet`. This saved me a bunch of keystrokes and typing out long paths. In the following example you can see (1) the full path to the `lists` directory and (2) the sym-link version:

- `/raid/datasets/imagenet/lists`
- `imagenet/lists` (which points to the full `/raid` path above)

To create your own sym-links (in a Unix-based environment) you can use the `ln` command. The example command below will create a symbolic link named `imagenet` in my current working directory which links to the full ImageNet dataset in my `/raid` drive:

```
$ ln -s /raid/datasets/imagenet imagenet
```

You can modify the above command to your own paths.

Regardless of where you choose to store your base ImageNet dataset directory, take the time now to create two subdirectories – `lists` and `rec`:

```
$ mkdir imagenet/lists
$ mkdir imagenet/rec
```

In the above command I assume you have created a sym-link named `imagenet` to point to the base dataset directory. If not, please specify your full path. The `lists` and `rec` subdirectories will be used later in this chapter.

The `build_imagenet.py` script will be responsible for building the mappings from input image file to output class label. The `train_alexnet.py` script will be used to train AlexNet from scratch on ImageNet. Finally, the `test_alexnet.py` script will be used to evaluate the performance of AlexNet on our test set.

The latter two scripts will be covered in Chapter 6, so for the time being, let's simply review `imagenet_alexnet_config.py` – this file will remain *largely unchanged* for all ImageNet experiments we run in this book. Therefore, it's important for us to take the time to understand how this file is structured.

Go ahead and open up `imagenet_alexnet_config.py` and insert the following code:

```
1  # import the necessary packages
2  from os import path
3
4  # define the base path to where the ImageNet dataset
5  # devkit are stored on disk)
6  BASE_PATH = "/raid/datasets/imagenet/ILSVRC2015"
```

Line 2 imports the only Python package we need, the `path` sub-module. The `path` sub-module contains a special variable named `path.sep` – this is the path separator for your operating system. On Unix machines, the path separator is `/` – an example file path may look like `path/to/your/file.txt`. However, on Windows the path separator is `\`, making the example file path `path\to\your\file.txt`. We would like our configuration to work agnostic of the operating system, so we'll use the `path.sep` variable whenever convenient.

Line 6 then defines the `BASE_PATH` to where our ImageNet dataset resides on disk. This directory should contain your four `Annotations`, `Data`, `devkit`, and `ImageSets` directories.

Due to the size of the ImageNet dataset, I decided to store all ImageNet related files on the RAID drive of system (which is why you see the `BASE_PATH` start with `/raid`). You should modify the `BASE_PATH` and update it to where the ImageNet dataset is stored on your system. Feel free to use the `datasets` directory that we have used in previous examples of *Deep Learning for Computer Vision with Python* – the `datasets` project structure will work just fine provided you have enough space on your main partition to store the entire dataset.

Again, I want to draw attention that you *will* need to update the `BASE_PATH` variable on your own system – please take the time to do that now.

From our `BASE_PATH` we can derive three more important paths:

```
8  # based on the base path, derive the images base path, image sets
9  # path, and devkit path
10 IMAGES_PATH = path.sep.join([BASE_PATH, "Data/CLS-LOC"])
11 IMAGE_SETS_PATH = path.sep.join([BASE_PATH, "ImageSets/CLS-LOC/"])
12 DEVKIT_PATH = path.sep.join([BASE_PATH, "devkit/data"])
```

The `IMAGES_PATH` is joined with the `BASE_PATH` to point to the directory that contains our *raw images* for the `test`, `train`, and `val` images. The `IMAGE_SETS_PATH` points to the directory containing the important `train_cls.txt` and `val.txt` files which *explicitly* list out the filenames for each set. Finally, as the name suggests, the `DEVKIT_PATH` is the base path to where our DevKit lives, in particular our plaintext files that we'll be parsing in Section 5.1.5.

Speaking of the DevKit, let's define `WORD_IDS`, the path to the `map_clsloc.txt` file which maps the 1,000 possible WordNet IDs to (1) the unique identifying integers and (2) human readable labels.

```
14 # define the path that maps the 1,000 possible WordNet IDs to the
15 # class label integers
16 WORD_IDS = path.sep.join([DEVKIT_PATH, "map_clsloc.txt"])
```

In order to build our training set, we need to define `TRAIN_LIST`, the path that contains the $\approx$ 1.2 million (partial) image filenames for the training data:

```
# define the paths to the training file that maps the (partial)
# image filename to integer class label
TRAIN_LIST = path.sep.join([IMAGE_SETS_PATH, "train_cls.txt"])
```

Next, we need to define some validation configurations:

```
# define the paths to to the validation filenames along with the
# file that contains the ground-truth validation labels
VAL_LIST = path.sep.join([IMAGE_SETS_PATH, "val.txt"])
VAL_LABELS = path.sep.join([DEVKIT_PATH,
     "ILSVRC2015_clsloc_validation_ground_truth.txt"])

# define the path to the validation files that are blacklisted
VAL_BLACKLIST = path.sep.join([DEVKIT_PATH,
     "ILSVRC2015_clsloc_validation_blacklist.txt"])
```

The `VAL_LIST` variable points to the `val.txt` file in the `ImageSets` directory. As a reminder, `val.txt` lists the (partial) image filenames for the 50,000 validation files. In order to obtain the ground-truth labels for the validation data, we need to define the `VAL_LABELS` path – doing so enable us to connect individual image file names with class labels. Finally, the `VAL_BLACKLIST` file contains the unique integer IDs of validation files that have been blacklisted. When we build the ImageNet dataset, we'll take explicit care to ensure these images are not included in the validation data.

In the next code block we define the `NUM_CLASSES` variable as well as `NUM_TEST_IMAGES`:

```
# since we do not have access to the testing data we need to
# take a number of images from the training data and use it instead
NUM_CLASSES = 1000
NUM_TEST_IMAGES = 50 * NUM_CLASSES
```

For the ImageNet dataset, there are 1,000 possible image classes, therefore, `NUM_CLASSES` is set to `1000`. In order to derive our testing set, we need to sample images from the training set. We'll set `NUM_TEST_IMAGES` to be $50 \times 1,000 = 50,000$ images. As I mentioned earlier, we'll be using the `im2rec` mxnet tool to convert our raw images files on disk to a record file suitable for training using the mxnet library.

To accomplish this action, we first need to define the `MX_OUTPUT` path and then derive a few other variables:

```
# define the path to the output training, validation, and testing
# lists
MX_OUTPUT = "/raid/datasets/imagenet"
TRAIN_MX_LIST = path.sep.join([MX_OUTPUT, "lists/train.lst"])
VAL_MX_LIST = path.sep.join([MX_OUTPUT, "lists/val.lst"])
TEST_MX_LIST = path.sep.join([MX_OUTPUT, "lists/test.lst"])
```

All files that are outputted either by (1) our Python helper utilities or (2) the `im2rec` binary will be stored in the base directory, `MX_OUTPUT`. Based on how I organize datasets (detailed in Chapter 6 of the *Starter Bundle*), I've chosen to include all output files in the `imagenet` directory, which also stores the raw images, DevKit, etc. You should store the output files wherever you feel comfortable – I am simply providing an example of how I organize datasets on my machine.

As I mentioned, after applying our Python utility scripts, we'll be left with three files – `train.lst`, `val.lst`, and `test.lst` – these files will contain the (integer) class label IDs and the full path to the image filenames for each of our data splits (**Lines 40-42**). The `im2rec` tool will then take these `.lst` files as input and create `.rec` files which store the actual raw images + class labels together, similar to building an HDF5 dataset in Chapter 10 of the *Practitioner Bundle*:

```
44 # define the path to the output training, validation, and testing
45 # image records
46 TRAIN_MX_REC = path.sep.join([MX_OUTPUT, "rec/train.rec"])
47 VAL_MX_REC = path.sep.join([MX_OUTPUT, "rec/val.rec"])
48 TEST_MX_REC = path.sep.join([MX_OUTPUT, "rec/test.rec"])
```

The difference here is that these record files are *much* more compact (as we can store images as compressed JPEG or PNG files, instead of raw NumPy array bitmaps). Furthermore, these record files are meant to be used *exclusively* with the mxnet library, allowing us to obtain better performance than the original HDF5 datasets.

I have chosen to include the `lists` and `rec` subdirectories inside the `imagenet` directory for organizational purposes – I would highly suggest you do the same. Provided that you follow my directory structure, please take the time to create your `lists` and `rec` subdirectories now. If you wait until we review and execute the `build_imagenet.py` you may forget to create these subdirectories, resulting in the script erroring out. But don't worry! You can simply go back create the `lists` and `rec` and re-execute the script.

When building our dataset, we'll need to compute the `DATASET_MEAN` for each of the RGB channels in order to perform mean normalization:

```
50 # define the path to the dataset mean
51 DATASET_MEAN = "output/imagenet_mean.json"
```

This configuration simply stores the path to where the means will be serialized to disk in JSON format. Provided you run all experiments on the same machine (or at least machines with identical ImageNet directory structures), the *only* configurations you will have to edit from experiment-to-experiment are the ones below:

```
53 # define the batch size and number of devices used for training
54 BATCH_SIZE = 128
55 NUM_DEVICES = 8
```

Line 54 defines the `BATCH_SIZE` in which images will be passed through the network during training. For AlexNet we'll use mini-batch sizes of `128`. Depending on how deep a given CNN is, we may want to decrease this batch size. The `NUM_DEVICES` attribute controls the number of devices (whether CPUs, GPUs, etc.) used when training a given neural network. You should configure this variable based on the number of devices you have available for training on your machine.

5.2.2 Our ImageNet Helper Utility

Now that we have created an example configuration file, let's move on to the `ImageNetHelper` class we'll use to generate the `.lst` files for the training, testing, and validation splits, respectively. This class is an important utility helper, so we'll update our `pyimagesearch` module and store it in a file named `imagenethelper.py` inside the `utils` submodule:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- io
|    |--- nn
|    |--- preprocessing
|    |--- utils
|    |    |--- __init__.py
|    |    |--- captchahelper.py
|    |    |--- imagenethelper.py
|    |    |--- ranked.py
```

Go ahead and open `imagenethelper.py` and we'll define the utility class:

```
1  # import the necessary packages
2  import numpy as np
3  import os
4
5  class ImageNetHelper:
6      def __init__(self, config):
7          # store the configuration object
8          self.config = config
9
10         # build the label mappings and validation blacklist
11         self.labelMappings = self.buildClassLabels()
12         self.valBlacklist = self.buildBlackist()
```

Line 6 defines the constructor to our `ImageNetHelper`. The constructor requires only a single parameter: an object named `config`. The `config` is actually the `imagenet_alexnet_config` file that we defined in the previous section. By passing in this file as an *object* to `ImageNetHelper`, we can access all of our file paths and additional configurations. We then build our label mappings and validation blacklist on **Lines 11 and 12**. We'll review both the `buildClassLabels` and `buildBlacklist` methods later in this section.

Let's start with `buildClassLabels`:

```
14     def buildClassLabels(self):
15         # load the contents of the file that maps the WordNet IDs
16         # to integers, then initialize the label mappings dictionary
17         rows = open(self.config.WORD_IDS).read().strip().split("\n")
18         labelMappings = {}
```

On **Line 17** we read the entire contents of the `WORD_IDs` file which maps the WordNet IDs to (1) the unique integer representing that class and (2) the human readable labels. We then define the `labelMappings` dictionary which take a WordNet ID as a key and the integer class label as the value.

Now that the entire `WORD_IDS` file has been loaded into memory, we can loop over each of the rows:

```
20          # loop over the labels
21          for row in rows:
22               # split the row into the WordNet ID, label integer, and
23               # human readable label
24               (wordID, label, hrLabel) = row.split(" ")
25
26               # update the label mappings dictionary using the word ID
27               # as the key and the label as the value, subtracting '1'
28               # from the label since MATLAB is one-indexed while Python
29               # is zero-indexed
30               labelMappings[wordID] = int(label) - 1
31
32          # return the label mappings dictionary
33          return labelMappings
```

For each `row`, we break it into a 3-tuple (since each entry in the row is separated by a space), consisting of:

1. The WordNet ID (`wordID`).
2. The unique *integer* class label ID (`label`).
3. The human readable name of the WordNet ID (`hrLabel`).

Now that we have these values we can update our `labelMappings` dictionary. The key to the dictionary is the WordNet ID, `wordID`. Our value is the `label` with a value of `1` subtracted from it.

Why do we subtract one? Keep in mind that the ImageNet tools provided by ILSVRC were built using *MATLAB*. The MATLAB programming language is one-indexed (meaning it starts counting from `1`) while the Python programming language is zero-indexed (we start counting from `0`). Therefore, to convert the MATLAB indexes to Python indexes, we simply subtract a value of `1` from the label.

The `labelMappings` dictionary is returned to the calling function on **Line 33**.

Next we have the `buildBlacklist` function:

```
35     def buildBlackist(self):
36          # load the list of blacklisted image IDs and convert them to
37          # a set
38          rows = open(self.config.VAL_BLACKLIST).read()
39          rows = set(rows.strip().split("\n"))
40
41          # return the blacklisted image IDs
42          return rows
```

This function is fairly straightforward. On **Line 38**, we read the entire contents of the `VAL_BLACKLIST` file. The `VAL_BLACKLIST` file contains the unique integer names of the *validation* files (one per line) that we should *exclude* from the validation set due to ambiguous labels. We simply break the string into a list (splitting on the newline `\n` character) and convert `rows` to a `set` object. A `set` object will allow us to determine if a given validation image is part of the blacklist in $O(1)$ time.

Our next function is responsible for ingesting the `TRAIN_LIST` and `IMAGES_PATH` configurations to construct a set of image paths and associated integer class labels for the training set:

```
44      def buildTrainingSet(self):
45          # load the contents of the training input file that lists
46          # the partial image ID and image number, then initialize
47          # the list of image paths and class labels
48          rows = open(self.config.TRAIN_LIST).read().strip()
49          rows = rows.split("\n")
50          paths = []
51          labels = []
```

On **Line 48** we load the entire contents of the `TRAIN_LIST` file and break it into rows on **Line 49**. Recall that the `TRAIN_LIST` file contains the *partial* image file paths – a sample of the `train_cls.txt` file can be seen below:

```
n01440764/n01440764_10969 91
n01440764/n01440764_10979 92
n01440764/n01440764_10995 93
n01440764/n01440764_11011 94
n01440764/n01440764_11018 95
n01440764/n01440764_11044 96
n01440764/n01440764_11063 97
n01440764/n01440764_11085 98
n01440764/n01440764_1108 99
n01440764/n01440764_1113 100
```

Our job will be to build both lists for the (1) the *full* image path and (2) corresponding class label (**Lines 50 and 51**). In order to build the lists, we need to loop over each or the `rows` individually:

```
53          # loop over the rows in the input training file
54          for row in rows:
55              # break the row into the partial path and image
56              # number (the image number is sequential and is
57              # essentially useless to us)
58              (partialPath, imageNum) = row.strip().split(" ")
59
60              # construct the full path to the training image, then
61              # grab the word ID from the path and use it to determine
62              # the integer class label
63              path = os.path.sep.join([self.config.IMAGES_PATH,
64                  "train", "{}.JPEG".format(partialPath)])
65              wordID = partialPath.split("/")[0]
66              label = self.labelMappings[wordID]
```

Each `row` consists of two entries – the `partialPath` to the training image file (e.x., `n01440764/n01440764_10026`) and the `imageNum`. The `imageNum` variable is simply a bookkeeping counter – it serves no purpose when building the training set; we'll be ignoring it. **Lines 63 and 64** are responsible for building the full `path` to the training image given the `IMAGES_PATH` and the `partialPath`.

This `path` consists of three components:

1. The `IMAGES_PATH` where all our `train`, `test`, and `val` directories live.
2. The hardcoded `train` string which indicates that we are constructing a file path for a training image.

3. The `partialPath` which is the *sub-directory* and *base filename* of the image itself.

We append the file extension to `.JPEG` to create the final image path. An example `path` can be seen below:

```
/raid/datasets/imagenet/ILSVRC2015/Data/CLS-LOC/train/n02097130/n02097130_4602.JPEG
```

When debugging your Python scripts used to generate these `.lst` files, *make sure you validate the file paths* before continuing. I would suggest using the simple `ls` command to check and see if the file exists. If `ls` comes back and tells you the file path doesn't exist, then you know you have an error in your configuration.

On **Line 65** we extract the `wordID`. The `wordID` is the sub-directory of the `partialPath`, therefore, we simply have to split on the `/` character and we can extract the `wordID`. Once we have the `wordID`, we can lookup the corresponding integer class label in `labelMappings` (**Line 66**).

Given the `path` and the `label`, we can update the `paths` and `labels` lists, respectively:

```
68                # update the respective paths and label lists
69                paths.append(path)
70                labels.append(label)
71
72            # return a tuple of image paths and associated integer class
73            # labels
74            return (np.array(paths), np.array(labels))
```

Line 74 returns a 2-tuple of the `paths` and `labels` to the calling function. These values will later be written to disk as a `.lst` file using our (to be defined) `build_dataset.py` script.

The final function we need to create our path is `buildValidationSet`, which is responsible for building our validation image paths and validation class labels:

```
76        def buildValidationSet(self):
77            # initialize the list of image paths and class labels
78            paths = []
79            labels = []
80
81            # load the contents of the file that lists the partial
82            # validation image filenames
83            valFilenames = open(self.config.VAL_LIST).read()
84            valFilenames = valFilenames.strip().split("\n")
85
86            # load the contents of the file that contains the *actual*
87            # ground-truth integer class labels for the validation set
88            valLabels = open(self.config.VAL_LABELS).read()
89            valLabels = valLabels.strip().split("\n")
```

Our `buildValidationSet` function is very similar to our `buildTrainingSet`, only with a few extra additions. To start, we initialize our list of image `paths` and class `labels` (**Lines 78 and 79**). We the load the contents of `VAL_LIST` which contains the partial filenames of the validation files (**Lines 83 and 84**). In order to build our class labels, we need to read in the contents of `VAL_LABELS` – this file contains the integer class label for each entry in `VAL_LIST`.

Given both `valFilenames` and `valLabels`, we can create our `paths` and `labels` lists:

```
91          # loop over the validation data
92          for (row, label) in zip(valFilenames, valLabels):
93               # break the row into the partial path and image number
94               (partialPath, imageNum) = row.strip().split(" ")
95
96               # if the image number is in the blacklist set then we
97               # should ignore this validation image
98               if imageNum in self.valBlacklist:
99                    continue
100
101              # construct the full path to the validation image, then
102              # update the respective paths and labels lists
103              path = os.path.sep.join([self.config.IMAGES_PATH, "val",
104                   "{}.JPEG".format(partialPath)])
105              paths.append(path)
106              labels.append(int(label) - 1)
107
108         # return a tuple of image paths and associated integer class
109         # labels
110         return (np.array(paths), np.array(labels))
```

On **Line 92** we loop over each of the `valFilenames` and `valLabels`. We unpack the `row` on **Line 94** to extract the `partialPath` along with the `imageNum`. Unlike in the training set, the `imageNum` is important here – we make a check on **Lines 98 and 99** to see if the `imageNum` is in the blacklist set, and if so, we ignore it.

From there, **Lines 103 and 104** construct the `path` to the validation file. We update the `paths` list on **Line 105**. The `labels` list is then updated on **Line 106** where we once again take care to subtract `1` from the `label` since it is zero indexed. Finally, **Line 110** returns the 2-tuple of validation paths and labels to the calling function.

Now that our `ImageNetHelper` is defined, we can move on to constructing the `.lst` files which will be fed into `im2rec`.

5.2.3 Creating List and Mean Files

Just like in our previous `build_*.py` scripts in previous chapters, the `build_imagenet.py` script will look very similar. At a high level, we will:

1. Build the training set.
2. Build the validation set.
3. Construct the testing set by sampling the training set.
4. Loop over each of the sets.
5. Write the image path + corresponding class label to disk.

Let's go ahead and start working on `build_imagenet.py` now:

```
1  # import the necessary packages
2  from config import imagenet_alexnet_config as config
3  from sklearn.model_selection import train_test_split
4  from pyimagesearch.utils import ImageNetHelper
5  import numpy as np
6  import progressbar
7  import json
8  import cv2
```

Line 2 imports our `imagenet_alexnet_config` module and aliases it as `config`. We then import the `train_test_split` function from scikit-learn so we can construct a testing split from our training set. We'll also import our newly defined `ImageNetHelper` class to aid us in building the imageNet dataset.

Next, we can build our training and validation paths + class labels:

```
10 # initialize the ImageNet helper and use it to construct the set of
11 # training and testing data
12 print("[INFO] loading image paths...")
13 inh = ImageNetHelper(config)
14 (trainPaths, trainLabels) = inh.buildTrainingSet()
15 (valPaths, valLabels) = inh.buildValidationSet()
```

We then need to sample `NUM_TEST_IMAGES` from `trainPaths` and `trainLabels` to construct our testing split:

```
17 # perform stratified sampling from the training set to construct a
18 # a testing set
19 print("[INFO] constructing splits...")
20 split = train_test_split(trainPaths, trainLabels,
21     test_size=config.NUM_TEST_IMAGES, stratify=trainLabels,
22     random_state=42)
23 (trainPaths, testPaths, trainLabels, testLabels) = split
```

From here, our code looks near identical to all our other previous "dataset building" scripts in *Deep Learning for Computer Vision with Python*:

```
25 # construct a list pairing the training, validation, and testing
26 # image paths along with their corresponding labels and output list
27 # files
28 datasets = [
29     ("train", trainPaths, trainLabels, config.TRAIN_MX_LIST),
30     ("val", valPaths, valLabels, config.VAL_MX_LIST),
31     ("test", testPaths, testLabels, config.TEST_MX_LIST)]
32
33 # initialize the list of Red, Green, and Blue channel averages
34 (R, G, B) = ([], [], [])
```

Lines 28-31 define a `datasets` list. Each entry in the `datasets` list is a 4-tuple, consisting of four values:

1. The type of split (i.e., training, testing, or validation).
2. The image paths.
3. The image labels.
4. The path to the output `.lst` file required by mxnet.

We'll also initialize the RGB channel averages on **Line 34**. Next, let's loop over each entry in the `datasets` list:

```
36 # loop over the dataset tuples
37 for (dType, paths, labels, outputPath) in datasets:
38     # open the output file for writing
```

```
39      print("[INFO] building {}...".format(outputPath))
40      f = open(outputPath, "w")
41
42      # initialize the progress bar
43      widgets = ["Building List: ", progressbar.Percentage(), " ",
44           progressbar.Bar(), " ", progressbar.ETA()]
45      pbar = progressbar.ProgressBar(maxval=len(paths),
46           widgets=widgets).start()
```

Line 40 opens a file pointer to our `outputPath`. We then build a `progressbar` widget on **Lines 43-46**. A progress bar is certainly not required, but I find it helpful to provide ETA information when building datasets (and computation could take while).

We now need to loop over each of the individual images and labels in the split:

```
48      # loop over each of the individual images + labels
49      for (i, (path, label)) in enumerate(zip(paths, labels)):
50           # write the image index, label, and output path to file
51           row = "\t".join([str(i), str(label), path])
52           f.write("{}\n".format(row))
53
54           # if we are building the training dataset, then compute the
55           # mean of each channel in the image, then update the
56           # respective lists
57           if dType == "train":
58                image = cv2.imread(path)
59                (b, g, r) = cv2.mean(image)[:3]
60                R.append(r)
61                G.append(g)
62                B.append(b)
63
64           # update the progress bar
65           pbar.update(i)
```

For each `path` and `label`, we write three values to the output `.lst` file:

1. The index, `i` (this is simply a unique integer that mxnet can associate with the image in the set).
2. The integer class `label`.
3. The *full* `path` to the image file.

Each of these values are separated by a tab, with one set of values per line. A sample of such an output file can be seen below:

```
0   35   /raid/datasets/imagenet/ILSVRC2015/Data/CLS-LOC/train/n02097130/n02097130_4602.JPEG
1   640  /raid/datasets/imagenet/ILSVRC2015/Data/CLS-LOC/train/n02276258/n02276258_7039.JPEG
2   375  /raid/datasets/imagenet/ILSVRC2015/Data/CLS-LOC/train/n03109150/n03109150_2152.JPEG
3   121  /raid/datasets/imagenet/ILSVRC2015/Data/CLS-LOC/train/n02483708/n02483708_3226.JPEG
4   977  /raid/datasets/imagenet/ILSVRC2015/Data/CLS-LOC/train/n04392985/n04392985_22306.JPEG
```

The `im2rec` tool in mxnet will then take these `.lst` files and build our `.rec` datasets. On **Lines 57-62** we check to see if our dataset type is `train` – and if so, we compute the RGB mean for the image and update the respective channel lists.

Our final code block handles cleaning up file pointers and serializing the RGB means to disk:

```
67      # close the output file
68      pbar.finish()
69      f.close()
70
71  # construct a dictionary of averages, then serialize the means to a
72  # JSON file
73  print("[INFO] serializing means...")
74  D = {"R": np.mean(R), "G": np.mean(G), "B": np.mean(B)}
75  f = open(config.DATASET_MEAN, "w")
76  f.write(json.dumps(D))
77  f.close()
```

To execute the `build_dataset.py` script, just execute the following command:

```
$ time python build_imagenet.py
[INFO] loading image paths...
[INFO] constructing splits...
[INFO] building /raid/datasets/imagenet/lists/train.lst...
Building List: 100% |####################################| Time: 1:47:35
[INFO] building /raid/datasets/imagenet/lists/val.lst...
Building List: 100% |####################################| Time: 0:00:00
[INFO] building /raid/datasets/imagenet/lists/test.lst...
Building List: 100% |####################################| Time: 0:00:00
[INFO] serializing means...

real    107m39.276s
user    75m26.044s
sys     2m35.237s
```

As you can see, the training set took the longest to complete at **107m39s** due to the fact that each of the 1.2 million images needed to be loaded from disk and have their means computed. The testing and validation `.lst` files were written to disk in a manner of seconds due to the fact that all we needed to do was write the image paths and labels to file (no extra I/O was required).

To validate that your `.lst` files for the training, testing, and validation files were successfully created, check the contents of your `MX_OUTPUT` directory. Below you can find the contents of my `MX_OUTPUT/lists` directory (where I choose to store the `.lst` files):

```
$ ls /raid/datasets/imagenet/lists/
test.lst  train.lst  val.lst
```

Doing a line count on each of the files reveals there are 50,000 images in our testing set, 50,000 images in our validation set, and 1,231,167 images in our testing set:

```
$ wc -l /raid/datasets/imagenet/lists/*.lst
    50000 /raid/datasets/imagenet/lists/test.lst
  1231167 /raid/datasets/imagenet/lists/train.lst
    48238 /raid/datasets/imagenet/lists/val.lst
  1329405 total
```

Given these `.lst` files, let's build our record packed dataset using mxnet's `im2rec`.

5.2.4 Building the Compact Record Files

Now that we have the `.lst` files, building a `.rec` file for each of the individual training, testing, and validation files is a breeze. Below you can find my example command to generate the `train.rec` file using mxnet's `im2rec` binary:

```
$ ~/mxnet/bin/im2rec /raid/datasets/imagenet/lists/train.lst "" \
    /raid/datasets/imagenet/rec/train.rec \ resize=256 encoding='.jpg' \
    quality=100
```

On my system, I compiled the `mxnet` library in my home directory; therefore, the full path to the `im2rec` is `~/mxnet/im2rec`. You may need to adjust this path if you compiled `mxnet` in a different location on your machine. Another alternative solution is to simply place the `im2rec` binary on your system `PATH`.

The first argument to `im2rec` is the path to our output `train.lst` file – this path should match the `TRAIN_MX_LIST` variable on `imagenet_alexnet_config.py`. The second argument is a blank empty string. This parameter indicates the root path to our image files. Since we have already derived the *full path* to the images on disk inside our `.lst` files, we can simply leave this parameter blank. We then have the final *required* parameter to `im2rec` – this is the path to our output record file database, where our compressed images and class labels will be stored. This parameter should match the `TRAIN_MX_REC` variable in `imagenet_alexnet_config.py`.

Next, we supply three optional arguments to `im2rec`:

1. `resize`: Here we indicate that our image should be resized to 256 pixels along the shortest dimension. This resizing not only reduces the input resolution of the image (thereby reducing the output `.rec` file size), but also enables us to perform data augmentation during training.
2. `encoding`: Here we can supply either JPEG or PNG encoding. We'll use JPEG encoding to save disk space as it's a lossy format. The PNG is a *lossless* format, but will result in much larger `.rec` files. Furthermore, we shouldn't be worried about a small amount of JPEG artifacts. JPEG artifacts exist in real-world images and can even be viewed as at type of data augmentation.
3. `quality`: This value is the *image quality* of either the JPEG or PNG image. For JPEG images, this value ranges from 1-100 (the larger the value, the less compression/artifacts introduced, but resulting in a larger file size). For PNG images, this value will range from 1-9. In this example, we'll supply a JPEG value of 100 to not further introduce additional JPEG artifacts in our dataset.

As the output below demonstrates, converting the `.lst` files into packed record files is time consuming as *each* of the $\approx 1.2 million$ images needs to be loaded from disk, resized, and stored in the dataset:

```
[17:25:21] tools/im2rec.cc:126: New Image Size: Short Edge 256
[17:25:21] tools/im2rec.cc:139: Encoding is .jpg
[17:25:21] tools/im2rec.cc:185: Write to output: /raid/datasets/imagenet/rec/train.rec
[17:25:21] tools/im2rec.cc:187: Output: /raid/datasets/imagenet/rec/train.rec
[17:25:21] tools/im2rec.cc:200: JPEG encoding quality: 100
[17:25:26] tools/im2rec.cc:295: 1000 images processed, 5.81205 sec elapsed
[17:25:33] tools/im2rec.cc:295: 2000 images processed, 12.213 sec elapsed
[17:25:39] tools/im2rec.cc:295: 3000 images processed, 18.6334 sec elapsed
...
[19:22:42] tools/im2rec.cc:295: 1230000 images processed, 7041.57 sec elapsed
[19:22:49] tools/im2rec.cc:295: 1231000 images processed, 7048.79 sec elapsed
[19:22:51] tools/im2rec.cc:298: Total: 1231167 images processed, 7050.07 sec elapsed
```

On my machine, this entire process took a little under two hours.
I can then repeat the process for the validation set:

```
$ ~/mxnet/bin/im2rec /raid/datasets/imagenet/lists/val.lst "" \
    /raid/datasets/imagenet/rec/val.rec resize=256 encoding='.jpg' \
    quality=100
[06:40:10] tools/im2rec.cc:126: New Image Size: Short Edge 256
[06:40:10] tools/im2rec.cc:139: Encoding is .jpg
[06:40:10] tools/im2rec.cc:185: Write to output: /raid/datasets/imagenet/rec/val.rec
[06:40:10] tools/im2rec.cc:187: Output: /raid/datasets/imagenet/rec/val.rec
[06:40:10] tools/im2rec.cc:200: JPEG encoding quality: 100
[06:40:15] tools/im2rec.cc:295: 1000 images processed, 5.9966 sec elapsed
[06:40:22] tools/im2rec.cc:295: 2000 images processed, 12.0281 sec elapsed
[06:40:27] tools/im2rec.cc:295: 3000 images processed, 17.2865 sec elapsed
...
[06:44:36] tools/im2rec.cc:295: 47000 images processed, 266.616 sec elapsed
[06:44:42] tools/im2rec.cc:295: 48000 images processed, 272.019 sec elapsed
[06:44:43] tools/im2rec.cc:298: Total: 48238 images processed, 273.292 sec elapsed
```

As well as the testing set:

```
$ ~/mxnet/bin/im2rec /raid/datasets/imagenet/lists/test.lst "" \
    /raid/datasets/imagenet/rec/test.rec resize=256 encoding='.jpg' \
    quality=100
[06:47:16] tools/im2rec.cc:139: Encoding is .jpg
[06:47:16] tools/im2rec.cc:185: Write to output: /raid/datasets/imagenet/rec/test.rec
[06:47:16] tools/im2rec.cc:187: Output: /raid/datasets/imagenet/rec/test.rec
[06:47:16] tools/im2rec.cc:200: JPEG encoding quality: 100
[06:47:22] tools/im2rec.cc:295: 1000 images processed, 6.32423 sec elapsed
[06:47:28] tools/im2rec.cc:295: 2000 images processed, 12.3095 sec elapsed
[06:47:35] tools/im2rec.cc:295: 3000 images processed, 19.0255 sec elapsed
...
[06:52:47] tools/im2rec.cc:295: 49000 images processed, 331.259 sec elapsed
[06:52:55] tools/im2rec.cc:295: 50000 images processed, 339.409 sec elapsed
[06:52:55] tools/im2rec.cc:298: Total: 50000 images processed, 339.409 sec elapsed
```

After we have generated the datasets using `im2rec` for the training, testing, and validation splits, let's take a look at the file sizes:

```
$ ls -l /raid/datasets/imagenet/rec/
total 105743748
-rw-rw-r-- 1 adrian adrian   4078794388 Dec 12  2016 test.rec
-rw-rw-r-- 1 adrian adrian 100250277132 Dec 12  2016 train.rec
-rw-rw-r-- 1 adrian adrian   3952497240 Dec 12  2016 val.rec
```

Here we can see the `train.rec` file is the largest, coming in at just over 100GB. The `test.rec` and `val.rec` files are approximately 4GB apiece. The benefit here is that we were able to *substantially compress* our entire ImageNet dataset into efficiently packed record files. Not only has the compression saved us disk space, but it will dramatically speed up the training process as well due to the fact that less I/O operations will need to be performed.

This approach is in contrast to the HDF5 approach where we needed to store the raw NumPy array bitmap for each image. If we were to construct training split for ImageNet using HDF5 (using $256 \times 256 \times 3$ images), the resulting file would be over 1.9TB, an increase of 1,831 percent! Because of this increase, it's *highly advisable* to use the mxnet `.rec` format when working with large datasets.

5.3 Summary

In this chapter, we learned how to prepare the ImageNet dataset. We started by exploring the directory structure of both the *raw images* in ImageNet, followed by the metafiles in the DevKit. From there, we created our first ImageNet Python configuration file – all experiments performed on ImageNet will be derived from this original configuration file. Here we supplied paths to the raw input images, metafiles, number of class labels, output mean RGB serializations, etc. Provided that you use the *same* machine for running all ImageNet experiments (or at least an identical directory structure across all machines you run experiments on), this configuration file will *rarely* (if ever) need to be updated.

After constructing our configuration file, we defined the `ImageNetHelper` class. This class includes four utilities that facilitate our ability to generate mxnet list (called `.lst` files). Each `.lst` file includes three values per row:

1. The unique integer ID of the image.
2. The class label of the image.
3. The full path to the image residing on disk.

Given a `.lst` file for each of the training, testing, and validation splits, we then applied the `im2rec` binary (provided by mxnet) to create efficiently backed record files for each split. These record files are super compact and highly efficient. As we'll see in our next chapter when we train AlexNet from scratch on ImageNet, we'll leverage these `.rec` files using a special `ImageRecordIter`. Similar to our `HDF5DatasetGenerator`, the `ImageRecordIter` class will enable us to:

1. Iterate over all images in a given `.rec` file in mini-batches.
2. Apply data augmentation to each image in every batch.

To see how all the pieces fit together so we can train a Convolutional Neural Network *from scratch* on ImageNet, proceed to the next chapter.

6. Training AlexNet on ImageNet

In our previous chapter we discussed the ImageNet dataset in detail; specifically, the directory structure of the dataset and the supporting meta files used provide class labels for each image. From there, we defined two sets of files:

1. A configuration file to allow us to easily create new experiments when training Convolutional Neural Networks on ImageNet.
2. A set of utility scripts to prepare the dataset for the conversion from raw images residing on disk to an efficiently packed mxnet record file.

Using the `im2rec` binary provided by mxnet, along with the `.lst` files we created using our utility scripts, we were able to generate record files for each of our training, testing, and validation sets. The beauty of this approach is that the `.rec` files only have to be generated *once* – we can *reuse* these record files for *any* ImageNet classification experiment we wish to perform.

Secondly, the configuration files themselves are also reusable. While we built our first configuration file with AlexNet in mind, the reality is that we'll be using the *same* configuration file for VGGNet, GoogLeNet, ResNet, and SqueezeNet as well – the *only* aspect of the configuration file that needs to be changed when training a new network on ImageNet are:

1. The name of the network architecture (which is embedded in the configuration filename).
2. The batch size.
3. The number of GPUs to train the network on (if applicable).

In this chapter, we are first going to implement the AlexNet architecture using the mxnet library. We've already implemented AlexNet once back in Chapter 10 of the *Practitioner Bundle* using Keras. As you'll see, there are many parallels between mxnet and Keras, making it *extremely straightforward* to port an implementation between the two libraries. From there, I'll demonstrate how to train AlexNet on the ImageNet dataset.

This chapter, and all other chapters in this bundle that demonstrate how to train a given network architecture on the ImageNet dataset, are treated as a cross between a *case study* and a *lab journal*. For each of the chapters in this book, I've run tens to hundreds of experiments to gather the respective results. I want to share with you my thought process when training deep, state-of-the-art neural networks on the challenging ImageNet dataset so you can gain experience by watching me

obtain sub-optimal results – and then *tweaking* a few parameters to boost my accuracy to replicate the state-of-the-art performance. Sharing the "story" of how the network was trained, and not just the final result, will help you in your own deep learning experiments. Watching others, and then learning by experience, is the optimal way to quickly master the techniques required to be successful working with large image datasets and deep learning.

6.1 Implementing AlexNet

The first step in training AlexNet on ImageNet is to implement the AlexNet architecture using the mxnet library. We have already reviewed Krizhevsky et al.'s [8] seminal architecture in Chapter 10 of the *Practitioner Bundle* where we trained AlexNet on the Kaggle Dogs vs. Cats challenge, so this network should not feel new and unfamiliar. That said, I have included Table 6.1 representing the structure of the network as a matter of completeness.

We will now implement this architecture using Python and mxnet. As a personal preference, I like to keep my mxnet CNN implementations separate from my Keras CNN implementations. Therefore, I have created a sub-module named `mxconv` inside the `nn` module of `pyimagesearch`:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- nn
|    |    |--- __init__.py
|    |    |--- conv
|    |    |--- mxconv
|    |--- preprocessing
|    |--- utils
```

All network architectures implemented using mxnet will live inside this sub-module (hence why the module name starts with the text `mx`). Create a new file named `mxalexnet.py` inside `mxconv` to store our implementation of the `MxAlexNet` class:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- nn
|    |    |--- __init__.py
|    |    |--- conv
|    |    |--- mxconv
|    |    |    |--- __init__.py
|    |    |    |--- mxalexnet.py
|    |--- preprocessing
|    |--- utils
```

From there, we can start implementing AlexNet:

```
1  # import the necessary packages
2  import mxnet as mx
3
4  class MxAlexNet:
5      @staticmethod
6      def build(classes):
```

Layer Type	**Output Size**	**Filter Size / Stride**
INPUT IMAGE	$227 \times 227 \times 3$	
CONV	$55 \times 55 \times 96$	$11 \times 11/4 \times 4, K = 96$
ACT	$55 \times 55 \times 96$	
BN	$55 \times 55 \times 96$	
POOL	$27 \times 27 \times 96$	$3 \times 3/2 \times 2$
DROPOUT	$27 \times 27 \times 96$	
CONV	$27 \times 27 \times 256$	$5 \times 5, K = 256$
ACT	$27 \times 27 \times 256$	
BN	$27 \times 27 \times 256$	
POOL	$13 \times 13 \times 256$	$3 \times 3/2 \times 2$
DROPOUT	$13 \times 13 \times 256$	
CONV	$13 \times 13 \times 384$	$3 \times 3, K = 384$
ACT	$13 \times 13 \times 384$	
BN	$13 \times 13 \times 384$	
CONV	$13 \times 13 \times 384$	$3 \times 3, K = 384$
ACT	$13 \times 13 \times 384$	
BN	$13 \times 13 \times 384$	
CONV	$13 \times 13 \times 256$	$3 \times 3, K = 256$
ACT	$13 \times 13 \times 256$	
BN	$13 \times 13 \times 256$	
POOL	$13 \times 13 \times 256$	$3 \times 3/2 \times 2$
DROPOUT	$6 \times 6 \times 256$	
FC	4096	
ACT	4096	
BN	4096	
DROPOUT	4096	
FC	4096	
ACT	4096	
BN	4096	
DROPOUT	4096	
FC	1000	
SOFTMAX	1000	

Table 6.1: A table summary of the AlexNet architecture. Output volume sizes are included for each layer, along with convolutional filter size/pool size when relevant.

```
7           # data input
8           data = mx.sym.Variable("data")
```

Line 2 imports our only required library, `mxnet` aliased as `mx` for convenience. For readers similar with OpenCV and the `cv2` library, all functionality is neatly organized and contained inside a single import – which has the benefit of keeping our import code block tidy and clean, but also requires the architecture code to be *slightly* more verbose.

We then define the `build` method to `MxAlexNet` on **Line 6**, a standard we have developed throughout the entirety of this book. The `build` method is responsible for constructing the network architecture and returning it to the calling function.

However, examining the structure, you'll notice that we only need a single argument, `classes`, the total number of class labels in our dataset. The number of arguments is in contrast to implementing CNNs in Keras where we also need to supply the spatial dimensions, including width, height, and depth. How come we don't need to supply these values to the mxnet library? As we'll find out, the reason mxnet does not explicitly require the spatial dimensions is because the `ImageRecordIter` class (the class responsible for reading images from our record files) *automatically infers* the spatial dimensions of our images – there is no need to explicitly pass the spatial dimensions into the class.

Line 8 defines a mxnet variable named `data` – this is a very important variable as it represents the *input data* to the neural network. Without this variable our network would not be able to receive inputs, thus we would not be able to train or evaluate it.

Next, let's implement the first set of `CONV => RELU => POOL` layers:

```
10          # Block #1: first CONV => RELU => POOL layer set
11          conv1_1 = mx.sym.Convolution(data=data, kernel=(11, 11),
12               stride=(4, 4), num_filter=96)
13          act1_1 = mx.sym.LeakyReLU(data=conv1_1, act_type="elu")
14          bn1_1 = mx.sym.BatchNorm(data=act1_1)
15          pool1 = mx.sym.Pooling(data=bn1_1, pool_type="max",
16               kernel=(3, 3), stride=(2, 2))
17          do1 = mx.sym.Dropout(data=pool1, p=0.25)
```

As you can see, the mxnet API is similar to that of Keras. The function names have changed slightly, but overall, it's easy to see how mxnet function names map to Keras function names (e.x., `Conv2D` in Keras is simply `Convolution` in mxnet). Each layer in mxnet requires that you pass in a `data` argument – this data argument is the *input* to the layer. Using Keras and the `Sequential` model, the input would be automatically inferred. Conversely, mxnet gives you the flexibility to easily build *graph structures* where the input to one layer isn't necessarily the output of the preceding layer. This flexibility is especially handy when we start defining more exotic architectures such as GoogLeNet and ResNet.

Lines 11 and 12 define our first `CONV` layer. This layer takes our input `data` as an input, then applies a `kernel` size of 11×11 pixels using a `stride` of 4×4 and learning `num_filter=96` filters.

The original implementation of AlexNet used standard ReLU layers; however, I've often found that ELUs perform better, especially on the ImageNet dataset; therefore, we'll apply an ELU on **Line 13**. The ELU activation is implemented inside the `LeakyReLU` class which accepts our first `CONV` layer, `conv1_1` as an input. We then supply `act_type="elu"` to indicate that we wish to use the ELU variant of the Leaky ReLU family. After applying the activation, we'll want to perform batch normalization on **Lines 14** via the `BatchNorm` class. Note that we do not have to supply

the channel axis in which to normalize the activations – the channel axis is determined by mxnet automatically.

To reduce the spatial dimensions of the input we can apply the `Pooling` class on **Lines 15 and 16**. The `Pooling` class accepts the output of `bn1_1` as its input, then applies max pooling with a kernel size of 3×3 and a stride of 2×2. Dropout is applied on **Line 17** on help reduce overfitting.

Given this is your first exposure to implementing a layer set in mxnet, I would recommend going back and re-reading this section once or twice more. There are obvious parallels between the naming conventions in mxnet and Keras, but make sure you understand them now, *especially* the `data` argument and how we must explicitly define the *input* of a current layer as the *output* of a previous layer.

The rest of the AlexNet implementation I'll explain in less tedious detail as:

1. You have already implemented AlexNet once before in the *Practitioner Bundle*.
2. The code itself is quite self-explanatory and explaining each line of code *ad nauseum* would become tedious.

Our next layer set consists of another block of `CONV => RELU => POOL` layers:

```
19          # Block #2: second CONV => RELU => POOL layer set
20          conv2_1 = mx.sym.Convolution(data=do1, kernel=(5, 5),
21               pad=(2, 2), num_filter=256)
22          act2_1 = mx.sym.LeakyReLU(data=conv2_1, act_type="elu")
23          bn2_1 = mx.sym.BatchNorm(data=act2_1)
24          pool2 = mx.sym.Pooling(data=bn2_1, pool_type="max",
25               kernel=(3, 3), stride=(2, 2))
26          do2 = mx.sym.Dropout(data=pool2, p=0.25)
```

Here our `CONV` layer learns 256 filters, each of size 5×5. However, unlike Keras which can automatically infer the amount of padding required (i.e., `padding="same"`), we need to explicitly supply the `pad` value, as detailed in the table above. Supplying `pad=(2, 2)` ensures that the input and output spatial dimensions are the same.

After the `CONV` layer, another ELU activation is applied, followed by a `BN`. Max pooling is applied to the output of `BN`, reducing the spatial input size down to 13×13 pixels. Again, dropout is applied to reduce overfitting.

To learn deeper, more rich features, we'll stack multiple `CONV => RELU` layers on top of each other:

```
28          # Block #3: (CONV => RELU) * 3 => POOL
29          conv3_1 = mx.sym.Convolution(data=do2, kernel=(3, 3),
30               pad=(1, 1), num_filter=384)
31          act3_1 = mx.sym.LeakyReLU(data=conv3_1, act_type="elu")
32          bn3_1 = mx.sym.BatchNorm(data=act3_1)
33          conv3_2 = mx.sym.Convolution(data=bn3_1, kernel=(3, 3),
34               pad=(1, 1), num_filter=384)
35          act3_2 = mx.sym.LeakyReLU(data=conv3_2, act_type="elu")
36          bn3_2 = mx.sym.BatchNorm(data=act3_2)
37          conv3_3 = mx.sym.Convolution(data=bn3_2, kernel=(3, 3),
38               pad=(1, 1), num_filter=256)
39          act3_3 = mx.sym.LeakyReLU(data=conv3_3, act_type="elu")
40          bn3_3 = mx.sym.BatchNorm(data=act3_3)
41          pool3 = mx.sym.Pooling(data=bn3_3, pool_type="max",
42               kernel=(3, 3), stride=(2, 2))
43          do3 = mx.sym.Dropout(data=pool3, p=0.25)
```

Our first CONV layer learns 384, 3×3 filters using a padding size of $(1, 1)$ to ensure the input spatial dimensions match the output spatial dimensions. An activation is applied immediately following the convolution, followed by batch normalization. Our second CONV layer also learns 384, 3×3 filters, again followed by an activation and a batch normalization. The final CONV in the layer set reduces the number of filters learned to 256; however, maintains the same file size of 3×3.

The output of the final CONV is then passed through an activation and a batch normalization. A POOL operation is once again used to reduce the spatial dimensions of the volume. Dropout follows the POOL to help reduce overfitting.

Following along with Table 6.1 above, the next step in implementing AlexNet is to define the two FC layer sets:

```
45          # Block #4: first set of FC => RELU layers
46          flatten = mx.sym.Flatten(data=do3)
47          fc1 = mx.sym.FullyConnected(data=flatten, num_hidden=4096)
48          act4_1 = mx.sym.LeakyReLU(data=fc1, act_type="elu")
49          bn4_1 = mx.sym.BatchNorm(data=act4_1)
50          do4 = mx.sym.Dropout(data=bn4_1, p=0.5)
51
52          # Block #5: second set of FC => RELU layers
53          fc2 = mx.sym.FullyConnected(data=do4, num_hidden=4096)
54          act5_1 = mx.sym.LeakyReLU(data=fc2, act_type="elu")
55          bn5_1 = mx.sym.BatchNorm(data=act5_1)
56          do5 = mx.sym.Dropout(data=bn5_1, p=0.5)
```

Each of the FC layers includes 4096 hidden units, each followed by an activation, batch normalization, and more aggressive dropout of 50%. It is common to use dropouts of 40-50% in the FC layers as that is where the CNN connections are most dense and overfitting is most likely to occur.

Finally, we apply our softmax classifier using the `supplied` number of classes:

```
58          # softmax classifier
59          fc3 = mx.sym.FullyConnected(data=do5, num_hidden=classes)
60          model = mx.sym.SoftmaxOutput(data=fc3, name="softmax")
61
62          # return the network architecture
63          return model
```

After reviewing this implementation, you might be surprised at how similar the mxnet library is to Keras. While not identical, the function names are very easy to match together, as are the parameters. Perhaps the only inconvenience is that we must now *explicitly* compute our padding rather than on relying on the automatic padding inference of Keras. Otherwise, implementing Convolutional Neural Networks with mxnet is just as easy as Keras.

6.2 Training AlexNet

Now that we have implemented the AlexNet architecture in mxnet, we need to define a driver script responsible for actually *training* the network. Similar to Keras, training a network with mxnet is fairly straightforward, although there are two key differences:

1. The mxnet training code is slightly more verbose due to the fact that we would like to leverage multiple GPUs (if possible).

2. The mxnet library doesn't provide a convenient method to plot our loss/accuracy over time and instead logs training progress to the terminal.

Therefore, we need to use the `logging` package of Python to capture this output and save it to disk. We then manually inspect the output logs of training progress as well as write Python utility scripts to parse the logs and plot our training/loss. It's slightly more tedious than using Keras; however, the benefit of being to train networks *substantially faster* due to (1) mxnet being a compiled C++ library with Python bindings and (2) multiple GPUs is well worth the tradeoff.

I have included an example of mxnet training log below:

```
Start training with [gpu(0), gpu(1), gpu(2), gpu(3), gpu(4),
     gpu(5), gpu(6), gpu(7)]
Epoch[0] Batch [1000] Speed: 1677.33 samples/sec Train-accuracy=0.004186
Epoch[0] Batch [1000] Speed: 1677.33 samples/sec Train-top_k_accuracy_5=0.0181
Epoch[0] Batch [1000] Speed: 1677.33 samples/sec Train-cross-entropy=6.748022
Epoch[0] Resetting Data Iterator
Epoch[0] Time cost=738.577
Saved checkpoint to "imagenet/checkpoints/alexnet-0001.params"
Epoch[0] Validation-accuracy=0.008219
Epoch[0] Validation-top_k_accuracy_5=0.031189
Epoch[0] Validation-cross-entropy=6.629663
Epoch[1] Batch [1000] Speed: 1676.29 samples/sec Train-accuracy=0.028924
Epoch[1] Batch [1000] Speed: 1676.29 samples/sec Train-top_k_accuracy_5=0.0967
Epoch[1] Batch [1000] Speed: 1676.29 samples/sec Train-cross-entropy=5.883830
Epoch[1] Resetting Data Iterator
Epoch[1] Time cost=734.455
Saved checkpoint to "imagenet/checkpoints/alexnet-0002.params"
Epoch[1] Validation-accuracy=0.052816
Epoch[1] Validation-top_k_accuracy_5=0.150838
Epoch[1] Validation-cross-entropy=5.592251
Epoch[2] Batch [1000] Speed: 1675.09 samples/sec Train-accuracy=0.073691
Epoch[2] Batch [1000] Speed: 1675.09 samples/sec Train-top_k_accuracy_5=0.2045
Epoch[2] Batch [1000] Speed: 1675.09 samples/sec Train-cross-entropy=5.177066
Epoch[2] Resetting Data Iterator
Epoch[2] Time cost=733.579
Saved checkpoint to "imagenet/checkpoints/alexnet-0003.params"
Epoch[2] Validation-accuracy=0.094177
Epoch[2] Validation-top_k_accuracy_5=0.240031
Epoch[2] Validation-cross-entropy=5.039742
```

Here you can see that I am training AlexNet using eight GPUs (one GPU is sufficient, but I used eight in order to gather results faster). After every set number of batches (which we'll define later), the training loss, rank-1, and rank-5 accuracy are logged to file. Once the epoch completes, the training data iterator is reset, a checkpoint file created and model weights serialized, and the validation loss, rank-1, and rank-5 accuracy displayed. As we can see, after the first epoch, AlexNet is obtaining $\approx 3\%$ rank-1 accuracy on the training data and $\approx 6\%$ rank-1 accuracy on the validation data.

6.2.1 What About Training Plots?

The training log is easy enough to read and interpret; however, scanning a plain text file does not make up for the lack of *visualization* – actually *visualizing* a plot of the loss and accuracy over time can enable us to make better, more informed decisions regarding whether we need to adjust the learning rate, apply more regularization, etc.

The mxnet library (unfortunately) does not ship out-of-the-box with a tool to parse the logs and construct a training plot, so I have created a separate Python tool to accomplish this task for us. Instead of further bloating this chapter with utility code (which simply amounts to using basic programming and regular expressions to parse the log), I have decided to cover the topic directly on the PyImageSearch blog – you can learn more about the `plot_log.py` script here:

http://pyimg.co/ycyao

However, for the time being simply understand that this script is used to parse mxnet training logs and plot our respective training and validation losses and accuracies.

6.2.2 Implementing the Training Script

Now that we have defined the AlexNet architecture, we need to create a Python script to actually *train* the network on the ImageNet dataset. To start, open up a new file, name it `train_alexnet.py`, and insert the following code:

```
1  # import the necessary packages
2  from config import imagenet_alexnet_config as config
3  from pyimagesearch.nn.mxconv import MxAlexNet
4  import mxnet as mx
5  import argparse
6  import logging
7  import json
8  import os
```

Lines 2-8 import our required Python packages. Notice how we're importing our `imagenet_alexnet_config`, aliased as `config`, so we can access our ImageNet-specific training configurations. We'll then import our implementation of the AlexNet architecture in mxnet on **Line 3**. As I mentioned earlier in this chapter, mxnet logs training progress to file; therefore, we need the `logging` package on **Line 6** to capture the output of mxnet and save it directly to a file that we can later parse.

From here, let's parse our command line arguments:

```
10 # construct the argument parse and parse the arguments
11 ap = argparse.ArgumentParser()
12 ap.add_argument("-c", "--checkpoints", required=True,
13     help="path to output checkpoint directory")
14 ap.add_argument("-p", "--prefix", required=True,
15     help="name of model prefix")
16 ap.add_argument("-s", "--start-epoch", type=int, default=0,
17     help="epoch to restart training at")
18 args = vars(ap.parse_args())
```

Our `train_alexnet.py` script requires two switches, followed by a third optional one. The `-checkpoints` switch controls the path to the output directory where our model weights will be serialized after *each* epoch. Unlike Keras where we need to explicitly define when a model will be serialized to disk, mxnet does so automatically after *every* epoch.

The `-prefix` command line argument the *name* of the architecture you are training. In our case, we'll be using a prefix of `alexnet`. The prefix name will be included in the filename of every serialized weights file.

Finally, we can also supply a `--start-epoch`. When training AlexNet on ImageNet, we'll inevitably notice signs of either training stagnation or overfitting. In this case we'll want to `ctrl +`

c out of the script, adjust our learning rate, and continue training. By supplying a `-start-epoch`, we can resume training from a *specific* previous epoch that has been serialized to disk.

When training our network *from scratch* we'll use the following command to kick off the training process:

```
$ python train_alexnet.py --checkpoints checkpoints --prefix alexnet
```

However, if we wanted to resume training from a *specific epoch* (in this case, epoch 50), we would provide this command:

```
$ python train_alexnet.py --checkpoints checkpoints --prefix alexnet \
    --start-epoch 50
```

Keep this process in mind when you are training AlexNet on the ImageNet dataset.

As I mentioned before, mxnet uses logging to display training progress. Instead of displaying the training log to `stdout`, we should instead capture the log and save it to disk so we can parse it and review it later:

```
20 # set the logging level and output file
21 logging.basicConfig(level=logging.DEBUG,
22     filename="training_{}.log".format(args["start_epoch"]),
23     filemode="w")
```

Lines 21 and 22 create a file named `training_{epoch}.log` based on the value of `--start-epoch`. Having a unique filename for each starting epoch will make it easier to plot accuracy and loss over time, as discussed in the PyImageSearch blog post above.

Next, we can load our RGB means from disk and compute the batch size:

```
25 # load the RGB means for the training set, then determine the batch
26 # size
27 means = json.loads(open(config.DATASET_MEAN).read())
28 batchSize = config.BATCH_SIZE * config.NUM_DEVICES
```

Previously, our batch size was always a fixed number since we were using only a single CPU/GPU/device to train our networks. However, now that we are starting to explore the world of *multiple GPUs*, our batch size is actually `BATCH_SIZE * NUM_DEVICES`. The reason we multiply our initial batch size by the total number of CPUs/GPUs/devices we are training our network with is because each device is parsing a separate batch of images *in parallel*. After all devices have processed their batch, mxnet updates the respective weights in the network. Therefore, our batch size actually *increases* with the number of devices we use for training due to parallelization.

Of course, we need to access our training data in order to train AlexNet on ImageNet:

```
30 # construct the training image iterator
31 trainIter = mx.io.ImageRecordIter(
32     path_imgrec=config.TRAIN_MX_REC,
33     data_shape=(3, 227, 227),
34     batch_size=batchSize,
35     rand_crop=True,
```

```
36          rand_mirror=True,
37          rotate=15,
38          max_shear_ratio=0.1,
39          mean_r=means["R"],
40          mean_g=means["G"],
41          mean_b=means["B"],
42          preprocess_threads=config.NUM_DEVICES * 2)
```

Lines 31-42 define the `ImageRecordIter` responsible for reading our batches of images from our training record file. Here we indicate that each *output image* from the iterator should be resized to 227×227 pixels while applying data augmentation, including random cropping, random marring, random rotation, and shearing. Mean subtraction is also applied inside the iterator as well.

In order to speed up training (and ensure our network is not waiting on new samples from the data iterator), we can supply a value for `preprocess_threads` – generates N threads that poll image batches for the iterator and apply data augmentation. I've found a good rule of thumb is to set the number of `preprocess_threads` to be double the number of devices you are using to train the network.

Just as we need to access our training data, we also need to access our validation data:

```
44  # construct the validation image iterator
45  valIter = mx.io.ImageRecordIter(
46          path_imgrec=config.VAL_MX_REC,
47          data_shape=(3, 227, 227),
48          batch_size=batchSize,
49          mean_r=means["R"],
50          mean_g=means["G"],
51          mean_b=means["B"])
```

The validation data iterator is identical to the training data iterator, with the exception that *no* data augmentation is being applied (but we do apply mean subtraction normalization).

Next, let's define our optimizer:

```
53  # initialize the optimizer
54  opt = mx.optimizer.SGD(learning_rate=1e-2, momentum=0.9, wd=0.0005,
55          rescale_grad=1.0 / batchSize)
```

Just like in the original Krizhevsky paper, we'll be training AlexNet using SGD with an initial learning rate of $1e-2$, a momentum term of $\gamma = 0.9$, and a L2-weight regularization (i.e., "weight decay) of 0.0005. The `rescale_grad` parameter in the SGD optimizer is *very important* as it scales the gradients by our batch size. Without this rescaling, our network may be unable to learn.

The next code block handles defining the path to our output checkpoints path directory, along with initializing the argument parameters and auxiliary parameters to the network:

```
57  # construct the checkpoints path, initialize the model argument and
58  # auxiliary parameters
59  checkpointsPath = os.path.sep.join([args["checkpoints"],
60          args["prefix"]])
61  argParams = None
62  auxParams = None
```

In the case that we are training AlexNet from the very first epoch, we need to build the network architecture:

```
64 # if there is no specific model starting epoch supplied, then
65 # initialize the network
66 if args["start_epoch"] <= 0:
67      # build the LeNet architecture
68      print("[INFO] building network...")
69      model = MxAlexNet.build(config.NUM_CLASSES)
```

Otherwise, if we are *restarting training* from a specific epoch, we need to load the serialized weights from disk and extract the argument parameters, auxiliary parameters, and model "symbol" (i.e., what mxnet calls a compiled network):

```
71 # otherwise, a specific checkpoint was supplied
72 else:
73      # load the checkpoint from disk
74      print("[INFO] loading epoch {}...".format(args["start_epoch"]))
75      model = mx.model.FeedForward.load(checkpointsPath,
76           args["start_epoch"])
77
78      # update the model and parameters
79      argParams = model.arg_params
80      auxParams = model.aux_params
81      model = model.symbol
```

Regardless of whether we are starting training from the first first epoch *or* we are restarting training from a specific epoch, we need to initialize the `FeedForward` object representing the network we wish to train:

```
83 # compile the model
84 model = mx.model.FeedForward(
85      ctx=[mx.gpu(1), mx.gpu(2), mx.gpu(3)],
86      symbol=model,
87      initializer=mx.initializer.Xavier(),
88      arg_params=argParams,
89      aux_params=auxParams,
90      optimizer=opt,
91      num_epoch=90,
92      begin_epoch=args["start_epoch"])
```

The `ctx` parameter controls the *context* of our training. Here we can supply a list of GPUs, CPUs, or devices used to train the network. In this case, I am using three GPUs to train AlexNet; however, you should modify this line based on the number of GPUs available on your system. If you have only one GPU, then **Line 85** would read:

```
85      ctx=[mx.gpu(0)],
```

The `initializer` controls the weight initialization method for all weight later in the network – here we are using Xavier (also known as Glorot) initialization, the default initialization method for

most Convolutional Neural Networks (and the one used by default for Keras). We'll allow AlexNet to train for a maximum of 100 epochs, but again, this value will likely be adjusted as you train your network and monitor the process – it could very well be *less* epochs or it could be *more* epochs.

Finally, the `begin_epochs` parameter controls which epoch we should resume training from. This parameter is important as it allows mxnet to keep its internal bookkeeping variables in order.

Just as Keras provides us with callbacks to monitor training performance, so does mxnet:

```
94 # initialize the callbacks and evaluation metrics
95 batchEndCBs = [mx.callback.Speedometer(batchSize, 500)]
96 epochEndCBs = [mx.callback.do_checkpoint(checkpointsPath)]
97 metrics = [mx.metric.Accuracy(), mx.metric.TopKAccuracy(top_k=5),
98       mx.metric.CrossEntropy()]
```

On **Line 95** we define a `Speedometer` callback that is called at the end of every *batch*. This callback will provide us with training information after every `batchSize * 500` batches. You may lower or raise this value depending on how frequently you would like training updates. A smaller value will result in *more* updates to the log, while a larger value will imply *less* updates to the log.

Line 96 defines our `do_checkpoint` callback that is called at the end of every *epoch*. This callback is responsible for serializing our model weights to disk. Again, at the end of every epoch, our network weights will be serialized to disk, enabling us to restart training from a specific epoch of need be.

Finally, **Lines 97 and 98** initialize our list of `metrics` callbacks. We'll be monitoring rank-1 accuracy (`Accuracy`), rank-5 accuracy (`TopKAccuracy`), as well as categorical cross-entropy loss.

All that's left to do now is train our network:

```
100 # train the network
101 print("[INFO] training network...")
102 model.fit(
103       X=trainIter,
104       eval_data=valIter,
105       eval_metric=metrics,
106       batch_end_callback=batchEndCBs,
107       epoch_end_callback=epochEndCBs)
```

A call to the `.fit` of the `model` starts (or restarts) the training process. Here we need to supply the training data iterator, validation iterator, and any callbacks. Once `.fit` is called, mxnet will start logging results to our output log file so we can review the training process.

While the code required to train a network using mxnet may seem slightly verbose, keep in mind that this is *literally a blueprint* for training *any* Convolutional Neural Network on the ImageNet dataset. When we implement and train other network architectures such as VGGNet, GoogLeNet, etc., we'll simply need to:

1. Change **Line 2** to properly set the configuration file.
2. Update the `data_shape` in the `trainIter` and `valIter` (only if the network requires different input image spatial dimensions).
3. Update the SGD optimizer on **Lines 54 and 55**.
4. Change the name of the model being initialized on **Line 69**.

Other than these three changes, there are ***literally no other updates*** required to our script when training various CNNs on the ImageNet dataset. I have purposely coded our trainings script to be *portable* and *extendible*. Future chapters in the *ImageNet Bundle* will require much less coding – we'll simply implement the new network architecture, make a few changes to the optimizer and

configuration file, and be up and running within a matter of minutes. My hope is that you'll use this same script when implementing and experimenting with your own deep learning network architectures.

6.3 Evaluating AlexNet

In this section, we'll learn how to evaluate a Convolutional Neural Network trained on the ImageNet dataset. This chapter specifically discusses AlexNet; however, as we'll see later in this book, the same script can be used to evaluate VGGNet, GoogLeNet, etc. as well, simply by changing the configuration import file.

To see how we can change the configuration, open up a new file, name it `test_alexnet.py`, and insert the following code:

```
1  # import the necessary packages
2  from config import imagenet_alexnet_config as config
3  import mxnet as mx
4  import argparse
5  import json
6  import os
```

Lines 2-6 import our required Python packages. Take special note of **Line 2** where we import our configuration file. As mentioned above, these *exact same* suite of scripts can be used to evaluate other networks simply by changing **Line 2** to match the configuration file for your respective network.

From there, we can parse our command line arguments:

```
8  # construct the argument parse and parse the arguments
9  ap = argparse.ArgumentParser()
10 ap.add_argument("-c", "--checkpoints", required=True,
11      help="path to output checkpoint directory")
12 ap.add_argument("-p", "--prefix", required=True,
13      help="name of model prefix")
14 ap.add_argument("-e", "--epoch", type=int, required=True,
15      help="epoch # to load")
16 args = vars(ap.parse_args())
```

Our script requires three switches, each of which are detailed below:

1. `-checkpoints`: This is the path to our output checkpoints directory during the training process.
2. `-prefix`: The prefix is the *name* of our actual CNN. When we run the `test_alexnet.py` script, we'll supply a value of `alexnet` for `-prefix`.
3. `-epoch`: Here we supply the epoch of our network that we wish to use for evaluation. For example, if we stopped our training after epoch 100, then we would use the 100th epoch for evaluating our network on the testing data.

These three command line arguments are required as they are all used to build the path to the serialized model weights residing on disk.

To evaluate our network, we need to create an `ImageRecordIter` to loop over the testing data:

```
18 # load the RGB means for the training set
19 means = json.loads(open(config.DATASET_MEAN).read())
```

```
20
21 # construct the testing image iterator
22 testIter = mx.io.ImageRecordIter(
23     path_imgrec=config.TEST_MX_REC,
24     data_shape=(3, 227, 227),
25     batch_size=config.BATCH_SIZE,
26     mean_r=means["R"],
27     mean_g=means["G"],
28     mean_b=means["B"])
```

Line 19 loads the average Red, Green, and Blue pixel values across the entire training set, exactly as we did during the training process. These values will be subtracted from each of the individual RGB channels in the image during testing *prior* to being fed through the network to obtain our output classifications. Recall that mean subtraction is a form of *data normalization* and thus needs to be performed on all three of the training, testing, and validation sets.

Lines 22-28 define the `testIter` used to loop over batches of images in the testing set. No data augmentation needs to be performed in this case, so we'll simply supply:

1. The path to the testing record file.
2. The intended spatial dimensions of the images (3 channels, with 227×227 width and height, respectively).
3. The batch size used during evaluation – this parameter is less important during testing as it is during trading as we are simply want to obtain our output predictions.
4. The RGB values used for mean subtraction/normalization.

Our AlexNet weights for each epoch are serialized to disk, so the next step is to define the path to the output checkpoints directory using the prefix (name of the network) and epoch (the *specific* weights that we wish to load):

```
30 # load the checkpoint from disk
31 print("[INFO] loading model...")
32 checkpointsPath = os.path.sep.join([args["checkpoints"],
33     args["prefix"]])
34 model = mx.model.FeedForward.load(checkpointsPath,
35     args["epoch"])
```

Lines 32 and 33 define the path to our output `-checkpoints` directory using the model `-prefix`. As we know during the training process, the output serialized weights are stored using the following filename convention:

```
checkpoints_directory/prefix-epoch.params
```

The `checkpointsPath` variable contains the `checkpoints_directory/prefix` portion of the file path. We then use **Lines 34 and 35** to:

1. Derive the rest of the file path using the supplied `-epoch` number.
2. Load the serialized parameters from disk.

Now that our pre-trained weights are loaded, we need to finish initializing the `model`:

```
37 # compile the model
38 model = mx.model.FeedForward(
39     ctx=[mx.gpu(0)],
```

```
40         symbol=model.symbol,
41         arg_params=model.arg_params,
42         aux_params=model.aux_params)
```

Line 38 defines the `model` as a `FeedForward` neural network. We'll use only a single GPU during evaluation (although you could certainly use more than one GPU or even your CPU). The argument parameters (`arg_params`) and auxiliary parameters (`aux_params`) are then set by accessing their respective values from the model loaded from disk.

Making predictions on the testing set is trivially easy:

```
44 # make predictions on the testing data
45 print("[INFO] predicting on test data...")
46 metrics = [mx.metric.Accuracy(), mx.metric.TopKAccuracy(top_k=5)]
47 (rank1, rank5) = model.score(testIter, eval_metric=metrics)
48
49 # display the rank-1 and rank-5 accuracies
50 print("[INFO] rank-1: {:.2f}%".format(rank1 * 100))
51 print("[INFO] rank-5: {:.2f}%".format(rank5 * 100))
```

Line 46 defines the list of `metrics` we are interested in – rank-1 and rank-5, respectively. We then make a call to the `.score` method of `model` to compute the rank-1 and rank-5 accuracies. The `.score` method requires that we pass in the `ImageRecordIter` object for the testing set, followed by the list of `metrics` that we wish to compute. Upon calling `.score` mxnet will loop over all batches of images in the testing set and compare these scores. Finally, **Lines 50 and 51** display the accuracies to our terminal.

At this point, we have all the ingredients we need to train AlexNet on the ImageNet dataset. We have:

1. A script to *train* the network.
2. A script to *plot* training loss and accuracy over time.
3. A script to *evaluate* the network.

The final step is to start running experiments and apply the scientific method to arrive at AlexNet model weights that replicate the performance of Krizhevsky et al.

6.4 AlexNet Experiments

When writing the chapters in this book, especially those related to training state-of-the-art network architectures on ImageNet, I wanted to provide *more* than just code and example results. Instead, I wanted to show the actual "story" of how a deep learning practitioner runs various experiments to obtain a desirable result. Thus, almost every chapter in the *ImageNet Bundle* contains a section like this one where I create a hybrid of lab journal meets case study. In the remainder of this section, I'll describe the experiments I ran, detail the results, and then describe the changes I made to improve the accuracy of AlexNet.

When evaluating and comparing AlexNet performance, we normally use the BVLC AlexNet implementation provided by Caffe [16] rather than the original AlexNet implementation. This comparison is due to a number of reasons, including different data augmentations being used by Krizhevsky et al. and the usage of the (now deprecated) Local Response Normalization layers (LRNs). Furthermore, the "CaffeNet" version of AlexNet tends to be more accessible to the scientific community. In the remainder of this section I'll be comparing my results to the CaffeNet benchmark, but still referring back to the original Krizhevsky et al. paper.

Epoch	Learning Rate
$1-50$	$1e-2$
$51-65$	$1e-3$
$66-80$	$1e-4$
$81-90$	$1e-5$

Table 6.2: Learning rate schedule used when training AlexNet on ImageNet for Experiment #1.

6.4.1 AlexNet: Experiment #1

In my first AlexNet on ImageNet experiment I decided to empirically demonstrate why we place batch normalization layers *after* the activation rather than *before* the activation. I also use standard ReLUs rather than ELUs to obtain a baseline for model performance (Krizhevsky et al. used ReLUs in their experiments). I thus modified the `mxalexnet.py` file detailed earlier in this chapter to reflect the batch normalization and activation changes, a sample of which can be seen below:

```
10          # Block #1: first CONV => RELU => POOL layer set
11          conv1_1 = mx.sym.Convolution(data=data, kernel=(11, 11),
12               stride=(4, 4), num_filter=96)
13          bn1_1 = mx.sym.BatchNorm(data=conv1_1)
14          act1_1 = mx.sym.Activation(data=bn1_1, act_type="relu")
15          pool1 = mx.sym.Pooling(data=act1_1, pool_type="max",
16               kernel=(3, 3), stride=(2, 2))
17          do1 = mx.sym.Dropout(data=pool1, p=0.25)
```

Notice how my batch normalization layer is now *before* the activation and I am using ReLU activation functions. I have included Table 6.2 to reflect my epoch number and associated learning rates below – we will review *why* I choose to lower the learning rates at each respective epoch in the remainder of this section.

I started training AlexNet using SGD with an initial learning rate of $1e-2$, a momentum term of 0.9, and L2 weight decay of 0.0005. The command to start the training process looked like this:

```
$ python train_alexnet.py --checkpoints checkpoints --prefix alexnet
```

I allowed my network to train, monitoring progress approximately every 10 epochs. One of the *worst* mistakes I see new deep learning practitioners make is checking their training plots *too often*. In most cases, you need the context of 10-15 epochs before you can make the decision that a network is indeed overfitting, underfitting, etc. After epoch 70, I plotted my training loss and accuracy (Figure 6.1), *top-left*). At this point, validation and training accuracy had essentially stagnated at $\approx 49-50\%$, a clear sign that the learning rate can be reduced to further improve accuracy.

Thus, I updated my learning rate to be $1e-3$ by editing **Lines 53 and 54** of `train_alexnet.py`:

```
53  # initialize the optimizer
54  opt = mx.optimizer.SGD(learning_rate=1e-3, momentum=0.9, wd=0.0005,
55       rescale_grad=1.0 / batchSize)
```

Notice how the learning rate has been decreased from $1e-2$ to $1e-3$, but all other SGD parameters have been left the same. I then restarted training from epoch 50 using the following command:

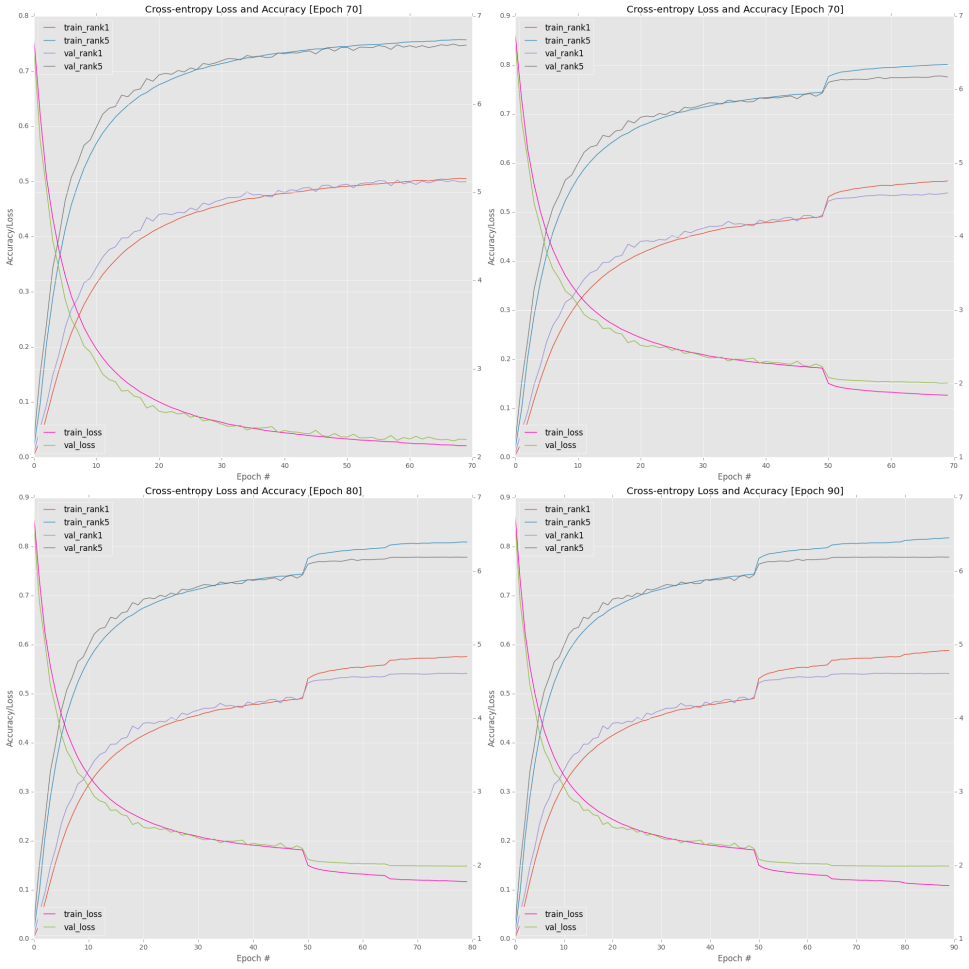


Figure 6.1: **Top-left:** Letting AlexNet train to epoch 70 with a $1e-2$ learning rate. Notice how rank-1 accuracy stagnates around $\approx 49\%$. I terminated training after epoch 70 and decided to restart training at epoch 50. **Top-right:** Restarting training from epoch 50 with a learning rate of $1e-3$. The order of magnitude decrease in α allows the network to "jump" to higher accuracy/lower loss. **Bottom-left:** Restarting training from epoch 65 with $\alpha = 1e-4$. **Bottom-right:** Epochs 80-90 at $\alpha = 1e-5$.

```
$ python train_alexnet.py --checkpoints checkpoints --prefix alexnet \
    --start-epoch 50
```

Again, I kept monitoring AlexNet progress until epoch 70 (Figure 6.1, *top-right*). The *first* key takeaway you should examine from this plot is how lowering my learning rate from $1e-2$ to $1e-3$ caused a *sharp rise* in accuracy and a *dramatic dip* in loss immediately past epoch 50 – this rise in accuracy and drop in loss is normal when you are training deep neural networks on large datasets. By lowering the learning rate we are allowing our network to descend into lower areas of loss, as previously the learning rate was too large for the optimizer to find these regions. Keep in mind that the goal of training a deep learning network is not necessarily to find a global minimum or even local minimum; rather to simply find a region where loss is sufficiently low.

However, toward the later epochs, I started to notice stagnation in the validation loss/accuracy (although the training accuracy/loss continued to improve). This stagnation tends to be a clear sign

that overfitting is starting to occur, but the gap between validation and training loss is *more* than acceptable, so I wasn't too worried. I updated my learning rate to be $1e-4$ (again, by editing **Lines 53 and 54** of `train_alexnet.py`) and restarted training from epoch 65:

```
$ python train_alexnet.py --checkpoints checkpoints --prefix alexnet \
    --start-epoch 65
```

Validation loss/accuracy improved *slightly*, but at this point, the learning rate is starting to become too small – furthermore, we are starting to overfit to the training data (Figure 6.1, *bottom-left*).

I finally allowed my network to train for 10 more epochs (80-90) using a $1e-5$ learning rate:

```
$ python train_alexnet.py --checkpoints checkpoints --prefix alexnet \
    --start-epoch 80
```

Figure 6.1 (*bottom-right*) right contains the resulting plot for the final ten epochs. Further training past epoch 90 is unnecessary as validation loss/accuracy has stopped improving while training loss continues to drop, putting us at risk at overfitting. At the end of epoch 90, I obtained 54.14% rank-1 accuracy and 77.90% rank-5 accuracy on the validation data. This accuracy is *very reasonable* for a first experiment, but not quite what I would expect from AlexNet-level performance, which the BVLC CaffeNet reference model reports to be approximately 57% rank-1 accuracy and 80% rank-5 accuracy.

I am purposely *not* evaluating my experiments on the testing data *yet*. I know there are more experiments to be run, and I try to only evaluate on the test set when I'm confident that I've obtained a high performing model. Remember that your testing set should be used *very sparingly* – you *do not* want to overfit to your testing set; otherwise, you'll completely destroy the ability for your model to generalize outside the samples in your dataset.

6.4.2 AlexNet: Experiment #2

The purpose of this experiment is to build on the previous one and demonstrate *why* we place batch normalization layers *after* the activation. I kept the ReLU activation, but swapped the ordering of the batch normalizations, as the following code block demonstrates:

```
10          # Block #1: first CONV => RELU => POOL layer set
11          conv1_1 = mx.sym.Convolution(data=data, kernel=(11, 11),
12               stride=(4, 4), num_filter=96)
13          act1_1 = mx.sym.Activation(data=conv1_1, act_type="relu")
14          bn1_1 = mx.sym.BatchNorm(data=act1_1)
15          pool1 = mx.sym.Pooling(data=bn1_1, pool_type="max",
16               kernel=(3, 3), stride=(2, 2))
17          do1 = mx.sym.Dropout(data=pool1, p=0.25)
```

Again, I used the exact same optimizer parameters of SGD with an initial learning rate of $1e-2$, momentum of 0.9, and L2 weight decay of 0.0005. Table 6.3 includes my epoch and associated learning rate schedule. I started training AlexNet using the following command:

```
$ python train_alexnet.py --checkpoints checkpoints --prefix alexnet
```

Epoch	Learning Rate
$1-65$	$1e-2$
$66-85$	$1e-3$
$86-100$	$1e-4$

Table 6.3: Learning rate schedule used when training AlexNet on ImageNet for Experiment #2 and Experiment #3.

Around epoch 65 I noticed that validation loss and accuracy were stagnating (Figure 6.2, *top-left*). Therefore, I stopped training, adjusted my learning rate to be $1e-3$, and then restarted training from the 65th epoch:

```
$ python train_alexnet.py --checkpoints checkpoints --prefix alexnet \
    --start-epoch 65
```

Again, we can see the characteristic jump in accuracy by lowering the learning rate, when validation accuracy/loss plateaus (Figure 6.2, *top-right*). At epoch 85 I again lowered my learning rate, this time from $1e-3$ to $1e-4$ and allowed the network to train for 15 more epochs, after which validation loss/accuracy stopped improving (Figure 6.2, *bottom*).

Examining the logs for my experiment, I noticed that my rank-1 accuracy was 56.72% and rank-5 accuracy was 79.62%, *much* better than my previous experiment of placing the batch normalization layer *before* the activation. Furthermore, these results are well within the statistical range of what true AlexNet-level performance looks like.

6.4.3 AlexNet: Experiment #3

Given that my previous experiment demonstrated placing batch normalization *after* the activation yielded better results, I decided to swap out the standard ReLU activations with ELU activations. In my experience, replacing ReLUs with ELUs can often add a 1-2% increase in your classification accuracy on the ImageNet dataset. Therefore, my `CONV => RELU` block now become:

```
10          # Block #1: first CONV => RELU => POOL layer set
11          conv1_1 = mx.sym.Convolution(data=data, kernel=(11, 11),
12              stride=(4, 4), num_filter=96)
13          act1_1 = mx.sym.LeakyReLU(data=conv1_1, act_type="elu")
14          bn1_1 = mx.sym.BatchNorm(data=act1_1)
15          pool1 = mx.sym.Pooling(data=bn1_1, pool_type="max",
16              kernel=(3, 3), stride=(2, 2))
17          do1 = mx.sym.Dropout(data=pool1, p=0.25)
```

Notice how the batch normalization layer is placed *after* the activation along with ELUs replacing ReLUs. During this experiment I used the exact same SGD optimizer parameters as my previous two trials. I also followed the same learning rate schedule from the second experiment (Table 6.3).

To replicate my experiment, you can use the following commands:

```
$ python train_alexnet.py --checkpoints checkpoints --prefix alexnet
...
$ python train_alexnet.py --checkpoints checkpoints --prefix alexnet \
```

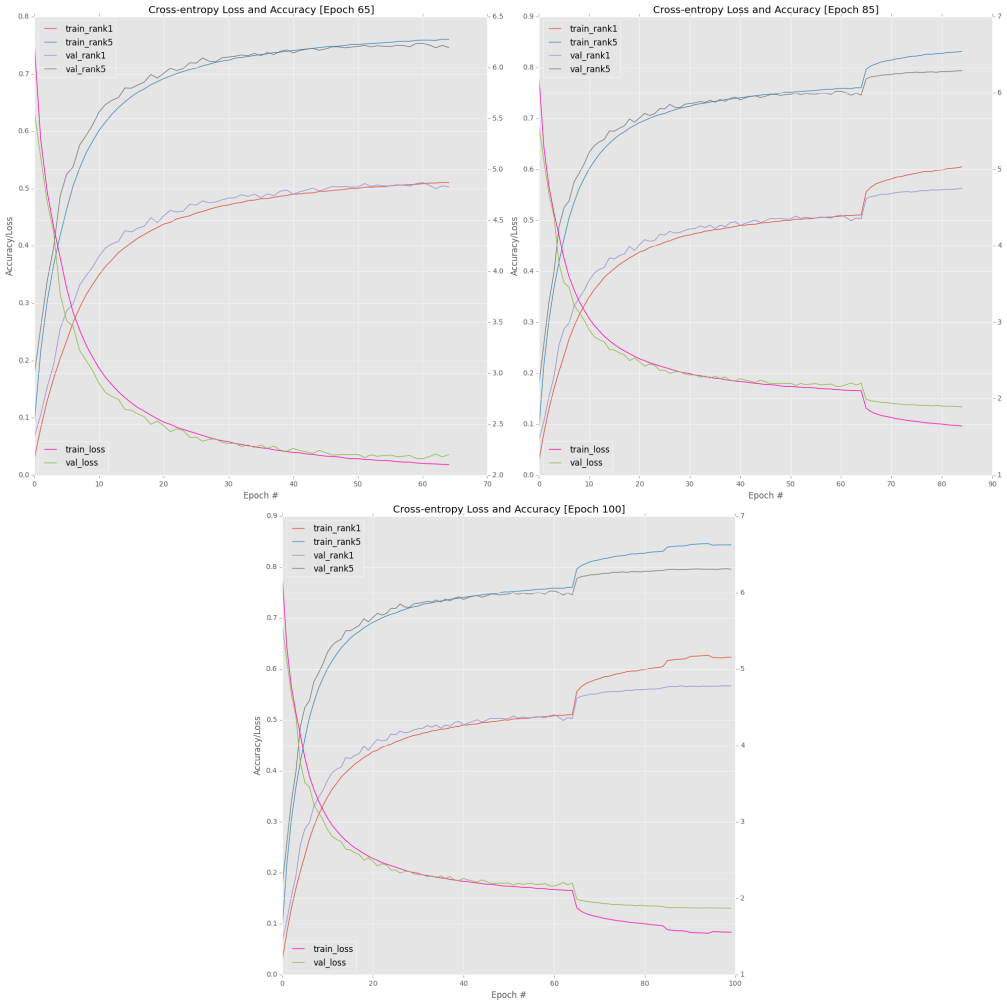


Figure 6.2: **Top-right:** The first 65 epochs with a $1e-2$ learning rate when placing the activation *before* the batch normalization. **Top-left**: Decreasing α to $1e-3$ causes a sharp increase in accuracy and decrease in loss; however, the training loss is decreasing significantly faster than validation loss. **Bottom:** A final decrease in α to $1e-4$ for epochs 85-100.

```
    --start-epoch 65
...
$ python train_alexnet.py --checkpoints checkpoints \
    --prefix alexnet --start-epoch 85
...
```

The first command starts training from the first epoch with an initial learning rate of $1e-2$. The second command restarts training at the 65 epoch using a learning rate of $1e-3$. And the final command restarts training at the 85th epoch with a learning rate of $1e-4$.

A full plot of the training and validation loss/accuracy can be seen in Figure 6.3. Again, you can see the clear characteristic marks of adjusting your learning rate by an order of magnitude at epochs 65 and 85, with the jumps becoming less pronounced as the learning rate decreases. I *did not* want to train past epoch 100 as AlexNet is clearly starting to overfit to the training data while validation accuracy/loss remains stagnate. The more this gap is allowed to grow, the worse overfitting becomes, therefore we apply the "early stopping" regularization criteria to prevent

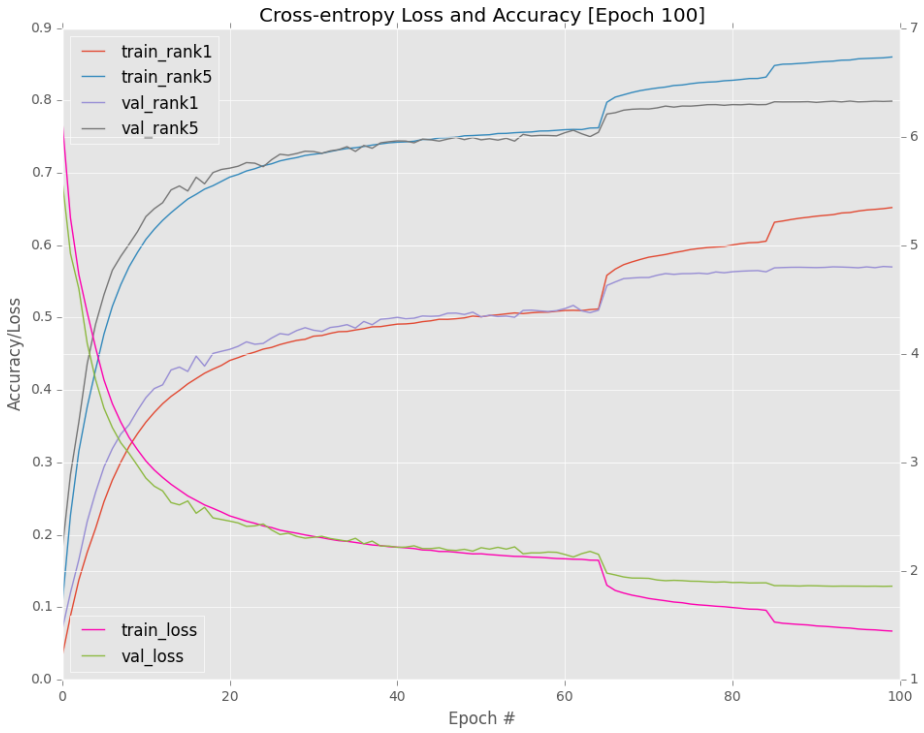


Figure 6.3: In our final AlexNet + ImageNet experiment we swap out ReLUs for ELUs and obtain a validation rank-1/rank5 accuracy of 57.00%/75.52% and testing rank-1/rank-5 accuracy of 59.80%/81.75%.

further overfitting.

Examining the accuracies for the 100th epoch, I found that I obtained 57.00% rank-1 accuracy and 79.52% rank-5 accuracy on the validation dataset. This result is only *marginally* better than my second experiment, but what's *very* interesting is what happens when I evaluated on the testing set using the `test_alexnet.py` script:

```
$ python test_alexnet.py --checkpoints checkpoints --prefix alexnet \
    --epoch 100
[INFO] loading model...
[INFO] predicting on test data...
[INFO] rank-1: 59.80%
[INFO] rank-5: 81.74%
```

I have summarized the results in Table 6.4. Here you can see that I obtained **59.80%** rank-1 and **81.75%** rank-5 accuracy on the testing set, certainly *above* what most independent papers and publications report AlexNet-level accuracy to be. For your convenience, I have included the weights for this AlexNet experiment in your download of the *ImageNet Bundle*.

Overall, the purpose of this section is to give you an idea of the types of experiments you'll need to run to obtain a reasonably performing model on the ImageNet dataset. Realistically, my lab journal included *25 separate experiments* for AlexNet + ImageNet, far too many to include in this book. Instead, I picked the ones most representative of important changes I made to the network architecture and optimizer. Keep in mind that for most deep learning problems you'll be running

	Testing Set
Rank-1 Accuracy	59.80%
Rank-5 Accuracy	81.75%

Table 6.4: Evaluating AlexNet on the ImageNet test set. Our results outperform the standard Caffe reference model used to benchmark AlexNet.

10-100 (and in some cases, even more) experiments before you obtain a model that performs well on both your validation and testing data.

Deep learning is not like other areas of programming where you write a function once and it works forever. Instead, there are many knobs and levers that need to be tweaked. Once you tweak the parameters, you'll be rewarded with a well performing CNN, but until then, be patient, ***and log your results!*** Making note of what *does* and *does not* work is *invaluable* – these notes will enable you to reflect on your experiments and identify new avenues to pursue.

6.5 Summary

In this chapter, we implemented the AlexNet architecture using the mxnet library and then trained it on the ImageNet dataset. This chapter was quite lengthy due to the need of us to comprehensively review the AlexNet architecture, the training script, and the evaluation script. Now that we have defined our training and evaluation Python scripts, we'll be able to *reuse them* in future experiments, making training and evaluation substantially easier – the main task will be for us to *implement* the actual network architecture.

The experiments we performed allowed us to identify two important takeaways:

1. Placing batch normalization *after* the activation (rather than *before*) will lead to higher classification accuracy/lower loss in most situations.
2. Swapping out ReLUs for ELUs can give you a small boost in classification accuracy.

Overall, we were able to obtain $\mathbf{59.80}\%$ rank-1 and $\mathbf{81.75}\%$ rank-5 accuracy on ImageNet, outperforming the standard Caffe reference model used to benchmark AlexNet.

7. Training VGGNet on ImageNet

In this chapter, we will learn how to train the VGG16 network architecture on the ImageNet dataset from scratch. The VGG family of Convolutional Neural Networks was first introduced by Simonyan and Zisserman in their 2014 paper, *Very Deep Convolutional Networks for Large Scale Image Recognition* [17].

This network is characterized by its simplicity, using only 3×3 convolutional layers stacked on top of each other in increasing depth. Reducing the spatial dimensions of volumes is accomplished through the usage of max pooling. Two fully-connected layers, each with 4,096 nodes (and dropout in between), are followed by a softmax classifier.

VGG is often used today for *transfer learning* as the network demonstrates an above average ability to generalize to datasets it was not trained on (as compared to other network types such as GoogLeNet and ResNet). More times than not, if you are reading a publication or lab journal that applies transfer learning, it likely uses VGG as the base model.

Unfortunately, training VGG from scratch is a pain, to say the least. The network is brutally slow to train, and the network architecture weights themselves are quite large (over 500MB). This is the *only* network inside the *ImageNet Bundle* that I would recommend that you *not* train if you do not have access to at least four GPUs. Due to the depth of the network along with the fully-connected layers, the backpropagation phase is excruciatingly slow.

In my case, training VGG on *eight* GPUs took $\approx$ 10 days – with any less than four GPUs, training VGG from scratch will likely take prohibitively long (unless you can be very patient). That said, it's important as a deep learning practitioner to understand the history of deep learning, especially the concept of *pre-training* and how we later learned to avoid this expensive operation by optimizing our initialization weight functions.

Again, this chapter is included in the *ImageNet Bundle* as the VGG family of networks is a *critical* aspect of deep learning; however, please do not feel the need to train this network from scratch – I have included the weights file derived from my experiment in this chapter for you to use in your own applications. Use this chapter as an educational reference so you can learn from VGG when you apply it to your own projects. In particular, this chapter will highlight the proper usage of the PReLU activation function and MSRA initialization.

7.1 Implementing VGGNet

When implementing VGG, Simonyan and Zisserman tried variants of VGG that increased in depth. Table 1 of their publication is included in Figure 7.1 below to highlight their experiments. In particular, we are most interested in configurations A, B, D, and E. You have already used both configuration D and E earlier in this book – these are the VGG16 and VGG19 architectures. Looking at these architectures, you'll notice two patterns:

The first is that the network uses *only* 3×3 filters. The second is as the depth of the network *increases*, the number of filters learned *increases* as well – to be exact, the number of filters *doubles* each time max pooling is applied to reduce volume size. The notion of doubling the number of filters each time you decrease spatial dimensions is of historical importance in the deep learning literature and even a pattern you will see today.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 7.1: A replication of Table 1 from Simonyan and Zisserman [17]. We'll be implementing configuration D in this chapter, commonly called VGG16.

The reason we perform this doubling of filters is to ensure no single layer block is more biased than the others. Layers earlier in the network architecture have *fewer filters*, but their spatial volumes are also *much larger*, implying there is "more (spatial) data" to learn from.

However, we know that applying a max pooling operation will reduce our spatial input volumes. If we reduce the spatial volumes *without* increasing the number of filters, then our layers become unbalanced and potentially biased, implying that layers earlier in the network may influence our output classification more than layers deeper in the network. To combat this imbalance, we keep in mind the ratio of volume size to number of filters. If we reduce the input volume size by 50-75%, then we double the number of filters in the next set of `CONV` layers to maintain the balance.

The issue with training such deep architectures is that Simonyan and Zisserman found training VGG16 and VGG19 to be extremely challenging due to their depth. If these architectures were

randomly initialized and trained from scratch, they would often struggle to learn and gain any initial "traction" – the networks were simply too deep for basic random initialization. Therefore, to train deeper variants of VGG, Simonyan and Zisserman came up with a clever concept called *pre-training*.

Pre-training is the practice of training *smaller* versions of your network architecture with fewer weight layers first and then using these converged network weights as the *initializations* for larger, deeper networks. In the case of VGG, the authors first trained configuration A, VGG11. VGG11 was able to converge to the level of reasonably low loss, but not state-of-the-art accuracy worthy.

The weights from VGG11 were then used as initializations to configuration B, VGG13. The `conv3-64` and `conv3-128` layers (highlighted in bold in Figure 7.1) in VGG13 were randomly initialized while the remainder of the layers were simply *copied over* from the pre-trained VGG11 network. Using the initializations, Simonyan and Zisserman were able to successfully train VGG13 – but still not obtain state-of-the-art accuracy.

This pre-training pattern continued to configuration D, which we commonly know as VGG16. This time *three* new layers were randomly initialized while the other layers were copied over from VGG13. The network was then trained using these "warmed pre-trained up" layers, thereby allowing the randomly initialized layers to converge and learn discriminating patterns. Ultimately, VGG16 was able to perform very well on the ImageNet classification challenge.

As a final experiment, Simonyan and Zisserman once again applied pre-training to configuration E, VGG19. This very deep architecture copied the weights from the pre-trained VGG16 architecture and then added another additional three convolutional layers. After training, it was found that VGG19 obtained the highest classification accuracy from their experiments; however, the size of the model (574MB) and the amount of time it took to train and evaluate the network, all for meager gains, made it less appealing to deep learning practitioners.

If pre-training sounds like a painful, tedious process, that's because it is. Training smaller variations of your network architecture and then using the converged weights as *initializations* to your deeper versions of the network is a clever trick; however, it requires training and tuning the hyperparameters to N separate networks, where N is your final network architecture along with the number of previous (smaller) networks required to obtain the end model. Performing this process is extremely time-consuming, especially for deeper networks with many fully-connected layers such as VGG.

The good news is that we no longer perform pre-training when training very deep Convolutional Neural Networks – instead, we rely on a good initialization function. Instead of pure random weight initializations we now use Xavier/Glorot [18] or MSRA (also known as He et al. initialization). Through the work of both Mishkin and Mtas in their 2015 paper, *All you need is a good init* [19] and He et al. in *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification* [20], we found that we can *skip* the pre-training phase entirely and jump *directly* to the deeper variations of the network architectures.

After these papers were published, Simonyan and Zisserman re-evaluated their experiments and found that these "smarter" initialization schemes and activation functions were able to replicate their previous performance *without* the usage of tedious pre-training.

Therefore, whenever you train VGG16 or VGG19 make sure you:

1. Swap out all ReLUs for PReLUs.
2. Use MSRA (also called "He et al. initialization") to initialize the weight layers in your network.

I additionally recommend using batch normalization *after* the activation functions in the network. Apply batch normalization was not discussed in the original Simonyan and Zisserman paper, but as other chapters have discussed, batch normalization can stabilize your training and reduce the total number of epochs required to obtain a reasonably performing model.

Now that we have learned about the VGG architecture, let's go ahead and implement configuration D from Simonyan and Zisserman, the seminal VGG16 architecture:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- nn
|    |    |--- __init__.py
|    |    |--- conv
|    |    |--- mxconv
|    |    |    |--- __init__.py
|    |    |    |--- mxalexnet.py
|    |    |    |--- mxvggnet.py
|    |--- preprocessing
|    |--- utils
```

You'll notice from the project structure above that I have created a file named `mxvggnet.py` in the `mxconv` sub-module of `pyimagesearch` – this file is where our implementation of VGG16 is going to live. Open up `mxvggnet.py` and we'll start to implement the network:

```
1  # import the necessary packages
2  import mxnet as mx
3
4  class MxVGGNet:
5      @staticmethod
6      def build(classes):
7          # data input
8          data = mx.sym.Variable("data")
```

Line 2 imports our `mxnet` library while **Line 4** defines the `MxVGGNet` class. Like all other CNNs we have implemented in this book, we'll create a `build` method on **Line 6** that is responsible for constructing the actual network architecture. This method accepts a single parameter, the number of `classes` that our network is supposed to discriminate amongst. **Line 8** then initializes the all important `data` variable, the actual input data to the CNN.

Looking at Figure 7.1 above, we can see our first block of layers should include `(CONV => RELU) * 2 => POOL`. Let's go ahead and define this layer block now:

```
10         # Block #1: (CONV => RELU) * 2 => POOL
11         conv1_1 = mx.sym.Convolution(data=data, kernel=(3, 3),
12             pad=(1, 1), num_filter=64, name="conv1_1")
13         act1_1 = mx.sym.LeakyReLU(data=conv1_1, act_type="prelu",
14             name="act1_1")
15         bn1_1 = mx.sym.BatchNorm(data=act1_1, name="bn1_1")
16         conv1_2 = mx.sym.Convolution(data=bn1_1, kernel=(3, 3),
17             pad=(1, 1), num_filter=64, name="conv1_2")
18         act1_2 = mx.sym.LeakyReLU(data=conv1_2, act_type="prelu",
19             name="act1_2")
20         bn1_2 = mx.sym.BatchNorm(data=act1_2, name="bn1_2")
21         pool1 = mx.sym.Pooling(data=bn1_2, pool_type="max",
22             kernel=(2, 2), stride=(2, 2), name="pool1")
23         do1 = mx.sym.Dropout(data=pool1, p=0.25)
```

The CONV layers in this block learn 64 filters, each of size 3×3. A leaky ReLU variant, PReLU (covered in Chapter 10 of the *Starter Bundle*) is used as our activation function. After every PReLU we then apply a batch normalization layer (not included in the original paper, but useful when stabilizing the training of VGG). At the end of the layer block a pooling layer is used to reduce the spatial dimensions of the volume using a 2×2 kernel and a 2×2 stride, reducing the volume size. We also apply dropout with a small percentage (25%) to help combat overfitting.

The second layer set in VGG16 also applies `(CONV => RELU) * 2 =< POOL`, only this time 128 filters are learned (again, each being 3×3). The number of filters has *doubled* since we have reduced the spatial input volume via the max pooling operation:

```
25          # Block #2: (CONV => RELU) * 2 => POOL
26          conv2_1 = mx.sym.Convolution(data=do1, kernel=(3, 3),
27               pad=(1, 1), num_filter=128, name="conv2_1")
28          act2_1 = mx.sym.LeakyReLU(data=conv2_1, act_type="prelu",
29               name="act2_1")
30          bn2_1 = mx.sym.BatchNorm(data=act2_1, name="bn2_1")
31          conv2_2 = mx.sym.Convolution(data=bn2_1, kernel=(3, 3),
32               pad=(1, 1), num_filter=128, name="conv2_2")
33          act2_2 = mx.sym.LeakyReLU(data=conv2_2, act_type="prelu",
34               name="act2_2")
35          bn2_2 = mx.sym.BatchNorm(data=act2_2, name="bn2_2")
36          pool2 = mx.sym.Pooling(data=bn2_2, pool_type="max",
37               kernel=(2, 2), stride=(2, 2), name="pool2")
38          do2 = mx.sym.Dropout(data=pool2, p=0.25)
```

Each CONV layer in this layer block is responsible of learning 128 filters. Again, the PReLU activation function is applied after every CONV layer, followed by a batch normalization layer. Max pooling is used to reduce the spatial dimensions of the volume followed by a small amount of dropout to reduce the effects of overfitting.

The third layer block of VGG adds an additional CONV layer, implying there are three total CONV operations prior to the max pooling:

```
40          # Block #3: (CONV => RELU) * 3 => POOL
41          conv3_1 = mx.sym.Convolution(data=do2, kernel=(3, 3),
42               pad=(1, 1), num_filter=256, name="conv3_1")
43          act3_1 = mx.sym.LeakyReLU(data=conv3_1, act_type="prelu",
44               name="act3_1")
45          bn3_1 = mx.sym.BatchNorm(data=act3_1, name="bn3_1")
46          conv3_2 = mx.sym.Convolution(data=bn3_1, kernel=(3, 3),
47               pad=(1, 1), num_filter=256, name="conv3_2")
48          act3_2 = mx.sym.LeakyReLU(data=conv3_2, act_type="prelu",
49               name="act3_2")
50          bn3_2 = mx.sym.BatchNorm(data=act3_2, name="bn3_2")
51          conv3_3 = mx.sym.Convolution(data=bn3_2, kernel=(3, 3),
52               pad=(1, 1), num_filter=256, name="conv3_3")
53          act3_3 = mx.sym.LeakyReLU(data=conv3_3, act_type="prelu",
54               name="act3_3")
55          bn3_3 = mx.sym.BatchNorm(data=act3_3, name="bn3_3")
56          pool3 = mx.sym.Pooling(data=bn3_3, pool_type="max",
57               kernel=(2, 2), stride=(2, 2), name="pool3")
58          do3 = mx.sym.Dropout(data=pool3, p=0.25)
```

Here the number of filters learned by each CONV layer jumps to 256, but the size of the filters remains at 3×3. The same pattern of PReLUs and batch normalization is used as well.

The fourth block of VGG16 also applies three CONV layers stacked on top of each other, but this time the total number of filters doubles again to 512:

```
60          # Block #4: (CONV => RELU) * 3 => POOL
61          conv4_1 = mx.sym.Convolution(data=do3, kernel=(3, 3),
62               pad=(1, 1), num_filter=512, name="conv4_1")
63          act4_1 = mx.sym.LeakyReLU(data=conv4_1, act_type="prelu",
64               name="act4_1")
65          bn4_1 = mx.sym.BatchNorm(data=act4_1, name="bn4_1")
66          conv4_2 = mx.sym.Convolution(data=bn4_1, kernel=(3, 3),
67               pad=(1, 1), num_filter=512, name="conv4_2")
68          act4_2 = mx.sym.LeakyReLU(data=conv4_2, act_type="prelu",
69               name="act4_2")
70          bn4_2 = mx.sym.BatchNorm(data=act4_2, name="bn4_2")
71          conv4_3 = mx.sym.Convolution(data=bn4_2, kernel=(3, 3),
72               pad=(1, 1), num_filter=512, name="conv4_3")
73          act4_3 = mx.sym.LeakyReLU(data=conv4_3, act_type="prelu",
74               name="act4_3")
75          bn4_3 = mx.sym.BatchNorm(data=act4_3, name="bn4_3")
76          pool4 = mx.sym.Pooling(data=bn4_3, pool_type="max",
77               kernel=(2, 2), stride=(2, 2), name="pool3")
78          do4 = mx.sym.Dropout(data=pool4, p=0.25)
```

Interestingly, the final set of CONV layers in VGG *does not* increase the number of filters and remains at 512. As to why this number did not double, I'm not entirely sure, but my best guess is either:

1. Jumping from 512 to 1024 filters introduced too many parameters in the network which caused VGG16 to overfit.
2. Training the network using 1024 filters in the final CONV blocks was simply too computationally expensive.

Regardless, the final set of CONV layers in VGG16 also learn 512 filters, each 3×3:

```
80          # Block #5: (CONV => RELU) * 3 => POOL
81          conv5_1 = mx.sym.Convolution(data=do4, kernel=(3, 3),
82               pad=(1, 1), num_filter=512, name="conv5_1")
83          act5_1 = mx.sym.LeakyReLU(data=conv5_1, act_type="prelu",
84               name="act5_1")
85          bn5_1 = mx.sym.BatchNorm(data=act5_1, name="bn5_1")
86          conv5_2 = mx.sym.Convolution(data=bn5_1, kernel=(3, 3),
87               pad=(1, 1), num_filter=512, name="conv5_2")
88          act5_2 = mx.sym.LeakyReLU(data=conv5_2, act_type="prelu",
89               name="act5_2")
90          bn5_2 = mx.sym.BatchNorm(data=act5_2, name="bn5_2")
91          conv5_3 = mx.sym.Convolution(data=bn5_2, kernel=(3, 3),
92               pad=(1, 1), num_filter=512, name="conv5_3")
93          act5_3 = mx.sym.LeakyReLU(data=conv5_3, act_type="prelu",
94               name="act5_3")
95          bn5_3 = mx.sym.BatchNorm(data=act5_3, name="bn5_3")
96          pool5 = mx.sym.Pooling(data=bn5_3, pool_type="max",
97               kernel=(2, 2), stride=(2, 2), name="pool5")
98          do5 = mx.sym.Dropout(data=pool5, p=0.25)
```

Our first set of FC layers includes 4,096 nodes and also follows the pattern of applying a PReLU and batch normalization:

```
100             # Block #6: FC => RELU layers
101             flatten = mx.sym.Flatten(data=do5, name="flatten")
102             fc1 = mx.sym.FullyConnected(data=flatten, num_hidden=4096,
103                  name="fc1")
104             act6_1 = mx.sym.LeakyReLU(data=fc1, act_type="prelu",
105                  name="act6_1")
106             bn6_1 = mx.sym.BatchNorm(data=act6_1, name="bn6_1")
107             do6 = mx.sym.Dropout(data=bn6_1, p=0.5)
```

Dropout is applied with a much stronger probability of 50% in the FC layers as the connections are much more dense here and prone to overfitting. Similarly, we add another FC layer set in the exact same manner below:

```
109             # Block #7: FC => RELU layers
110             fc2 = mx.sym.FullyConnected(data=do6, num_hidden=4096,
111                  name="fc2")
112             act7_1 = mx.sym.LeakyReLU(data=fc2, act_type="prelu",
113                  name="act7_1")
114             bn7_1 = mx.sym.BatchNorm(data=act7_1, name="bn7_1")
115             do7 = mx.sym.Dropout(data=bn7_1, p=0.5)
```

The final code block in VGG16 constructs FC layer for the total number of `classes` and then applies a softmax classifier:

```
117             # softmax classifier
118             fc3 = mx.sym.FullyConnected(data=do7, num_hidden=classes,
119                  name="fc3")
120             model = mx.sym.SoftmaxOutput(data=fc3, name="softmax")
121
122             # return the network architecture
123             return model
```

While there is certainly a lot more code required to implement VGG16 (as compared to AlexNet), it's all fairly straightforward. Furthermore, using a blueprint such as Figure 7.1 above makes the process *substantially easier*. When you implement your own sequential network architectures like these, I suggest writing the first set of CONV layers, then copying and pasting them for the next block, making sure you:

1. Change the inputs of each layer (i.e., the `data` argument) appropriately.
2. Increase the number of filters in the CONV layers according to your blueprint.

Doing so will help reduce any bugs that might be injected into your code due to writing out each layer definition by hand.

7.2 Training VGGNet

Now that we have implemented VGG16, we can train it on the ImageNet dataset. But first, let's define our project structure:

```
--- mx_imagenet_vggnet
|    |--- config
|    |    |--- __init__.py
|    |    |--- imagnet_vggnet_config.py
|    |--- output/
|    |--- test_vggnet.py
|    |--- train_vggnet.py
```

The project structure is essentially identical to the AlexNet structure in the previous chapter. We require a script named `train_vggnet.py` to actually train the network. The `test_vggnet.py` script will then be responsible for evaluating VGG16 on ImageNet. Finally, the `imagenet_vggnet_config.py` file contains our configurations for the experiment.

When defining this project structure, I simply *copied* the entire `mx_imagenet_alexnet` directory and then renamed the files to say `vggnet` instead of `alexnet`. Comparing `imagenet_vggnet_config.py` to `imagenet_alexnet_config.py`, you'll notice the configurations are *identical* (as the paths to our ImageNet dataset files do not change), with one exception:

```
53 # define the batch size and number of devices used for training
54 BATCH_SIZE = 32
55 NUM_DEVICES = 8
```

Here I have reduced the `BATCH_SIZE` for 128 down to 32. Due the depth and size of VGG16 (and therefore the amount of memory it consumes on our GPU), we won't be able to pass as many image batches through the network at one time. In order to fit VGG16 on my GPU (Titan X, 12GB), the batch size had to be reduced to 32. If you opt to train VGG16 from scratch on your own GPU, you might have to decrease the `BATCH_SIZE` further if you don't have as much GPU memory.

Secondly, I have also updated the number of GPUs I'll be using to train VGGNet to eight. As I mentioned at the beginning of this chapter, VGG16 is the *only* chapter in the ImageNet Bundle that I do not recommend training from scratch *unless* you have a four to eight GPUs. Even with this number of GPUs, it can easily take you 10-20 days to train the network.

Now that we have updated our configuration file, let's also update `train_vggnet.py`. Again, recall from Chapter 6 on AlexNet that we coded `train_alexnet.py` such that it requires *minimal changes* whenever training a new network architecture on ImageNet. As a matter of completeness I will review the entire file while highlighting the updates we have made; however, for a more exhaustive review of the training script, please see Chapter 6 where the entire template is reviewed in detail.

```
1 # import the necessary packages
2 from config import imagenet_vggnet_config as config
3 from pyimagesearch.nn.mxconv import MxVGGNet
4 import mxnet as mx
5 import argparse
6 import logging
7 import json
8 import os
```

On **Line 2** we import our `imagenet_vggnet_config` file from our project structure – this configuration file has been changed from the `imagenet_alexnet_config` import from the previous chapter. We'll also import `MxVGGNet`, our implementation of the VGG16 architecture. These are the only two changes required to our import section.

Next, let's parse our command line arguments and create our log file so mxnet can log training progress to it:

```
10 # construct the argument parse and parse the arguments
11 ap = argparse.ArgumentParser()
12 ap.add_argument("-c", "--checkpoints", required=True,
13      help="path to output checkpoint directory")
14 ap.add_argument("-p", "--prefix", required=True,
15      help="name of model prefix")
16 ap.add_argument("-s", "--start-epoch", type=int, default=0,
17      help="epoch to restart training at")
18 args = vars(ap.parse_args())
19
20 # set the logging level and output file
21 logging.basicConfig(level=logging.DEBUG,
22      filename="training_{}.log".format(args["start_epoch"]),
23      filemode="w")
24
25 # load the RGB means for the training set, then determine the batch
26 # size
27 means = json.loads(open(config.DATASET_MEAN).read())
28 batchSize = config.BATCH_SIZE * config.NUM_DEVICES
```

Line 27 loads the RGB means so we can apply mean subtraction data normalization while **Line 28** computes our `batchSize` based on the total number of devices we are using to train VGG16. Next, we need to construct the training data iterator:

```
30 # construct the training image iterator
31 trainIter = mx.io.ImageRecordIter(
32      path_imgrec=config.TRAIN_MX_REC,
33      data_shape=(3, 224, 224),
34      batch_size=batchSize,
35      rand_crop=True,
36      rand_mirror=True,
37      rotate=15,
38      max_shear_ratio=0.1,
39      mean_r=means["R"],
40      mean_g=means["G"],
41      mean_b=means["B"],
42      preprocess_threads=config.NUM_DEVICES * 2)
```

As well as the validation data iterator:

```
44 # construct the validation image iterator
45 valIter = mx.io.ImageRecordIter(
46      path_imgrec=config.VAL_MX_REC,
47      data_shape=(3, 224, 224),
48      batch_size=batchSize,
49      mean_r=means["R"],
50      mean_g=means["G"],
51      mean_b=means["B"])
```

We are now ready to initialize our optimizer:

```
53 # initialize the optimizer
54 opt = mx.optimizer.SGD(learning_rate=1e-2, momentum=0.9, wd=0.0005,
55      rescale_grad=1.0 / batchSize)
```

Following the paper by Simonyan and Zisserman, we'll be using the SGD optimizer with an initial learning rate of $1e-2$, a momentum term of 0.9, and a L2 weight decay of 0.0005. Special care is taken to rescale the gradients based on our batch size.

From there, we can construct the path to our `checkpointsPath` directory where the model weights will be serialized after every epoch:

```
57 # construct the checkpoints path, initialize the model argument and
58 # auxiliary parameters
59 checkpointsPath = os.path.sep.join([args["checkpoints"],
60      args["prefix"]])
61 argParams = None
62 auxParams = None
```

Next, we can determine if we are (1) training our model from the very first epoch or (2) restarting training from a *specific* epoch:

```
64 # if there is no specific model starting epoch supplied, then
65 # initialize the network
66 if args["start_epoch"] <= 0:
67      # build the LeNet architecture
68      print("[INFO] building network...")
69      model = MxVGGNet.build(config.NUM_CLASSES)
70
71 # otherwise, a specific checkpoint was supplied
72 else:
73      # load the checkpoint from disk
74      print("[INFO] loading epoch {}...".format(args["start_epoch"]))
75      model = mx.model.FeedForward.load(checkpointsPath,
76           args["start_epoch"])
77
78      # update the model and parameters
79      argParams = model.arg_params
80      auxParams = model.aux_params
81      model = model.symbol
```

Lines 66-69 handle if we are training VGG16 with no prior checkpoint. In this case, we instantiate the `MxVGGNet` class on **Line 69**. Otherwise, **Lines 72-81** assume we are loading a specific checkpoint from disk and restarting training (presumably after adjusting the learning rate to the SGD optimizer).

Finally, we can compile the model:

```
83 # compile the model
84 model = mx.model.FeedForward(
85      ctx=[mx.gpu(i) for i in range(0, config.NUM_DEVICES)],
```

```
86        symbol=model,
87        initializer=mx.initializer.MSRAPrelu(),
88        arg_params=argParams,
89        aux_params=auxParams,
90        optimizer=opt,
91        num_epoch=80,
92        begin_epoch=args["start_epoch"])
```

Our `model` will be trained with `ctx` of `NUM_DEVICES` GPUs – you should change this line based on the number of GPUs you are using on your system. Also, pay special attention to **Line 87** where we define the `initializer` – we'll be using `MSRAPrelu` as our initialization method, the exact same method suggested by He et al. [20] to train very deep neural networks. Without this initialization method, VGG16 would struggle to converge during training.

Next, let's define our set of callbacks and evaluation metrics:

```
94  # initialize the callbacks and evaluation metrics
95  batchEndCBs = [mx.callback.Speedometer(batchSize, 250)]
96  epochEndCBs = [mx.callback.do_checkpoint(checkpointsPath)]
97  metrics = [mx.metric.Accuracy(), mx.metric.TopKAccuracy(top_k=5),
98        mx.metric.CrossEntropy()]
```

Since VGG16 is a very slow network to train, I prefer to see *more* training updates logged to file; therefore, I decrease the number of batch size `Speedometer` updates (previously 500 for AlexNet) to log training updates *more frequently* to the log. We'll then monitor rank-1 accuracy, rank-5 accuracy, and categorical cross-entropy.

Our final code block handles training VGG16:

```
100 # train the network
101 print("[INFO] training network...")
102 model.fit(
103       X=trainIter,
104       eval_data=valIter,
105       eval_metric=metrics,
106       batch_end_callback=batchEndCBs,
107       epoch_end_callback=epochEndCBs)
```

As you can see from the implementation of `train_vggnet.py`, there are *very few differences* between the code detailed here and the `trian_alexnet.py` from the previous chapter – this similarity is ***exactly why*** I use this same template when training my own CNNs on ImageNet. I can simply copy the file, adjust a few import statements, instantiate my network, and then optionally adjust my optimizer and initializer parameters. Once you have done this process a few times, you can spin up a new ImageNet project in less than ten minutes.

7.3 Evaluating VGGNet

To evaluate VGGNet, we'll be using the `test_vggnet.py` script mentioned in our project structure above. Please note that this script is *identical* to `test_alexnet.py` in the previous chapter. There were *no changes* made to the script as the `test_*.py` scripts in this chapter are meant to be *templates* that can be applied and reapplied to any CNN trained on ImageNet.

Since the code is identical, I will not be reviewing `test_vggnet.py` here. Please consult Chapter 6 on `test_alexnet.py` for a thorough review of the code. Additionally, you can use the code downloads portion of this book to inspect the project and see the contents of `test_vggnet.py`. Again, the contents of these files are *identical* as they are part of our framework for training and evaluating CNNs trained on ImageNet.

7.4 VGGNet Experiments

When training VGG16, it was *critical* that I considered the experiments run by other researchers including Simonyan and Zisserman [17], He et al. [21, 22], and Mishkin et al. [19]. Through these works I was able to avoid running additional, expensive experiments and applied the following guidelines:

1. Skip pre-training in favor of better initialization methods.
2. Use MSRA/He et al. initialization.
3. Use PReLU activation functions.

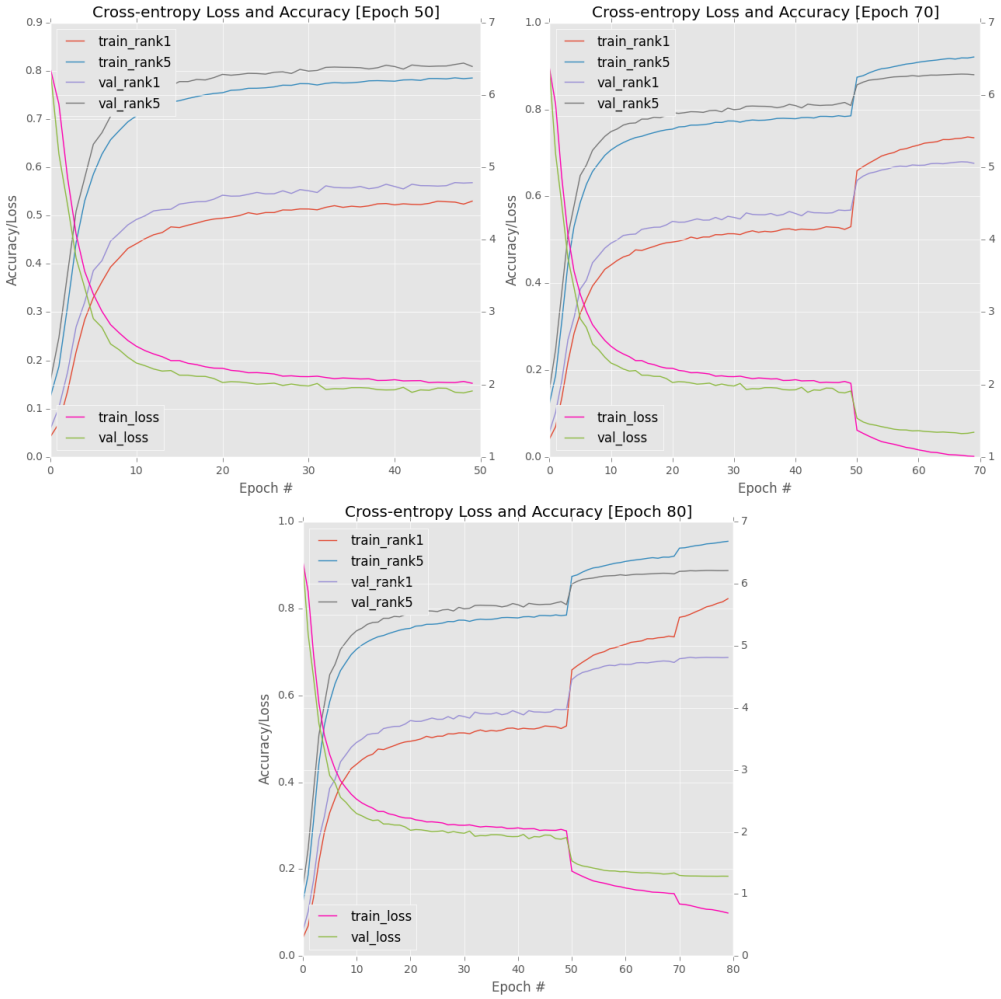


Figure 7.2: **Top-left:** Letting VGGNet train to epoch 50 with a $1e-2$ learning rate. **Top-right:** Restarting training from epoch 50 with a learning rate of $1e-2$. The order of magnitude decrease in α allows the network to "jump" to higher accuracy/lower loss. **Bottom:** Restarting training from epoch 70 with $\alpha = 1e-4$ for a total of 10 more epochs.

Because I followed these guides, I only needed to perform *one* experiment with VGG16 to replicate the results of Simonyan and Zisserman – this *single experiment* replicated their results near identically. In this experiment, I used the SGD optimizer to train VGG16 with an initial learning rate of $1e-2$, a momentum term of 0.9, and a L2 weight regularization of 0.0005. To speed up training, I used an Amazon EC2 instance with eight GPUs. I would *not* recommend trying to train VGG on a machine with less than four GPUs *unless* you are very patient.

I kicked off the VGG training process using the following command:

```
$ python train_vggnet.py --checkpoints checkpoints --prefix vggnet
```

I allowed the network to train until epoch 50 where both training and validation accuracy seemed to stagnate (Figure 7.2, *top-left*). I then `ctrl + c` out of the `train_vggnet.py` script and lowered the learning rate from $1e-2$ to $1e-3$:

```
53 # initialize the optimizer
54 opt = mx.optimizer.SGD(learning_rate=1e-3, momentum=0.9, wd=0.0005,
55      rescale_grad=1.0 / batchSize)
```

Training was then resumed using the following command:

```
$ python train_vggnet.py --checkpoints checkpoints  --prefix vggnet \
     --start-epoch 50
```

In the Figure 7.2 (*top-right*) you can see the results of lowering the learning rate over the course of 20 epochs. *Immediately* you can see the massive jump in both training and validation accuracy. Training and validation loss also fall with the learning rate update, as common when performing order of magnitude shifts when accuracy/loss saturates on deep Convolutional Neural Networks trained on large datasets, such as ImageNet.

Past epoch 70 I once again noted validation loss/accuracy stagnation while training loss continued to drop – this indicator shows overfitting starting to happen. In fact, we can see this divergence start to appear past epoch 60 in the accuracy plot: training accuracy continues to climb while validation accuracy remains the same.

In order to milk every last bit of performance out of VGG that I could (without overfitting too terribly), I once again dropped the learning rate from $1e-3$ to $1e-4$ and restarted training on epoch 70:

```
$ python train_vggnet.py --checkpoints checkpoints --prefix vggnet \
     --start-epoch 70
```

I then allowed the network to continue training for another 10 epochs until epoch 80 where I applied early stopping criteria (Figure 7.2, *bottom*). At this point, validation accuracy/loss had stagnated while the gap between training loss and validation loss started to diverge heavily, indicating that we are slightly overfitting and further training would only hurt the ability of the model to generalize. At the end of the 80th epoch, VGG16 was obtaining 68.77% rank-1 and 88.78% rank-5 validation accuracy. I then evaluated the 80th epoch on the test set using the following command:

Epoch	Learning Rate
$1-50$	$1e-2$
$51-70$	$1e-3$
$71-80$	$1e-4$

Table 7.1: Learning rate schedule used when training VGGNet on ImageNet.

```
$ python test_vggnet.py --checkpoints checkpoints --prefix vggnet \
    --epoch 80
[INFO] loading model...
[INFO] predicting on test data...
[INFO] rank-1: 71.42%
[INFO] rank-5: 90.03%
```

As my output demonstrates, VGG16 reached **71.42**% rank-1 and **90.03**% rank-5 accuracy, where is *nearly identical* to the original VGGNet paper by Simonyan and Zisserman. As a matter of completeness, I have included my learning rate schedule in the Table 7.1 for readers who wish to replicate these results.

The biggest downside to VGG16 (besides how long it takes to train) is the resulting model size, weighing in at over 533MB. If you were building a deep learning application and intended to *ship* the model size with your app, you would *already* have a package $> 500MB$ to distribute. Furthermore, all software is updated at some point, requiring you to repackage and re-distribute a large 500MB package, likely over a network connection. For high-speed broadband connections, this large model size may not be an issue. But for resource constrained devices such as embedded devices, cell phones, and even self-driving cars, this 500MB model size can be a *huge* burden. In these types of situations, we prefer very small model sizes.

Luckily, all remaining models we'll discuss in this bundle are substantially smaller than VGGNet. The weights for our *highly accurate* ResNet model come in at 102MB. GoogLeNet is even smaller at 28MB. And the super tiny, efficient SqueezeNet model size is only 4.9MB, making it ideal for any type of resource constrained deep learning.

7.5 Summary

In this chapter, we implemented the VGG16 architecture using the mxnet library and trained it from scratch on the ImageNet dataset. Instead of using the tedious, time-consuming process of pre-training to train smaller versions of our network architecture and subsequently using these pre-trained weights as initializations to our deeper architecture, we instead skipped this step, relying on the work of He et al. and Mishkin et al.:

1. We replaced standard ReLU activations with PReLUs.
2. We swapped Glorot/Xavier weight initialization for MSRA/He et al. initialization.

This process enabled us to replicate the work of Simonyan and Zisserman in a *single experiment*. Whenever training VGG-like architectures from scratch, definitely consider using PReLUs and MSRA initialization whenever possible. In some network architectures you won't notice an impact on performance when using PReLU + MSRA, but with VGG, the impact is *substantial*.

Overall, our version of VGG16 obtained **71.42**% rank-1 and **90.03**% rank-5 accuracy on the ImageNet test set, the highest accuracy we have seen thus far in this bundle. Furthermore, the VGG architecture has demonstrated itself to be well suited for generalization tasks. In our next chapter, we'll explore *micro-architectures* in more detail, including the GoogLeNet model, which will set the stage for more specialized micro-architectures including ResNet and SqueezeNet.

8. Training GoogLeNet on ImageNet

In this chapter we are going to implement the *full* GoogLeNet architecture introduced by Szgedy et al. in their 2014 paper, *Going deeper with convolutions* [23]. Both GoogLeNet and VGGNet were the top performers in the ImageNet Large Scale Visual Recognition challenge (ILSVRC) in 2014 with GoogLeNet *slightly* edging out VGGNet for the top #1 position. GoogLeNet also has the added benefit of being *significantly smaller* than VGG16 and VGG19 with only a 28.12MB model size versus the VGG, coming in at over 500MB.

That said, it has been shown that the VGG family of networks:

1. Achieve higher classification accuracy (in practice).
2. Generalize better.

The generalization, in particular, is why VGG is more often used in transfer learning problems such as feature extraction and *especially* fine-tuning. Later incarnations of GoogLeNet (which simply go by the name of Inception *N* where *N* is the version number released by Google) extend on the original GoogLeNet implementation and Inception module, bringing further accuracy – that said, we still tend to use VGG for transfer learning.

Finally, it's also worth mentioning that many researchers (myself included) have had trouble replicating the original results by Szegedy et al. Independent ImageNet experiments, such as the leaderboard maintained by vlfeat (an open source library for computer vision and machine learning), report both VGG16 and VGG19 outperforming GoogLeNet by a significant margin [24]. It is unclear exactly why this is, but as we'll find out in the remainder of this chapter, our GoogLeNet implementation does not outperform VGG, which is what we would expect from the original publication and ILSVRC 2014 challenge. Regardless, it's still important for us to study this seminal architecture and how to train it on the ImageNet dataset.

8.1 Understanding GoogLeNet

We'll start off this section with a quick review of the Inception module, the novel contribution by Szegedy et al. in their seminal work. From there we'll review the *full* GoogLeNet architecture that was used in the ILSVRC 2014 challenge. Finally, we'll implement GoogLeNet using Python and mxnet.

8.1.1 The Inception Module

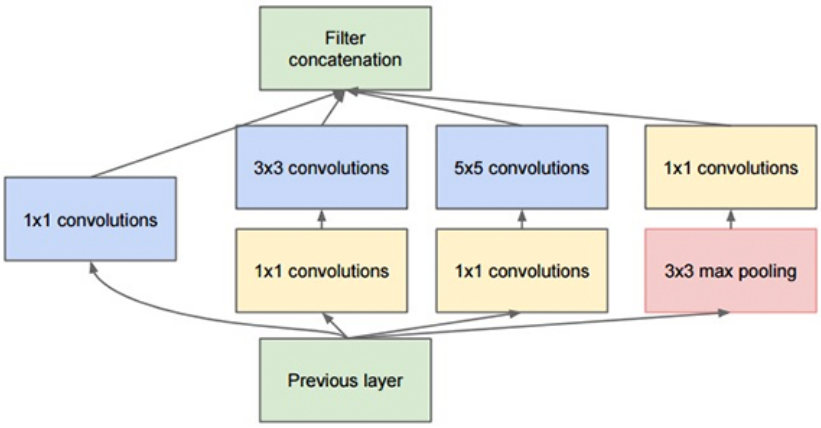


Figure 8.1: The original Inception module used in GoogLeNet. The Inception module acts as a "multi-level feature extractor" by computing 1×1, 3×3, and 5×5 convolutions within the *same* module of the network. Figure from Szegedy et al., 2014 [23].

The Inception module is a four branch micro-architecture used inside the GoogLeNet architecture (Figure 8.1). The primary purpose of the Inception module is to learn multi-scale features (1×1, 3×3, and 5×5 filters), then let the network "decide" which weights are the most important based on the optimization algorithm.

The *first* branch of the Inception module consists of entirely 1×1 filters. The *second* branch applies 1×1 convolutions followed by 3×3 filters. A *smaller* number of 1×1 filters are learned as a form of dimensionality reduction, thereby reducing the number of parameters in the overall architecture. The *third* branch is identical to the second branch, only instead of learning 3×3 filters, it instead learns 5×5 filters.

The *fourth* and final branch in the Inception module is called the *pool projection* branch. The pool projection applies 3×3 max pooling followed by a series of 1×1 convolutions. The reasoning behind this branch is that state-of-the-art Convolutional Neural Networks circa 2014 applied heavy usage of max pooling. It was postulated that to achieve high accuracy on the challenging ImageNet dataset, a network architecture *must* apply max pooling, hence why we see it inside the Inception module. We now know that max pooling is *not* a requirement in a network architecture and we can instead reduce volume size strictly through `CONV` layers [21, 25]; however, this was the prevailing thought at the time.

The output of the four branches are concatenated along the channel dimension, forming a stack of filters, and then passed into the next layer in the network. For a more detailed review of the Inception module, please refer to Chapter 12 of the *Practitioner Bundle*.

8.1.2 GoogLeNet Architecture

Figure 8.2 details the GoogLeNet architecture we will be implementing, including the number of filters for each of the four branches of the Inception module. After every convolution, it is *implicitly implied* (i.e., not shown in the table to save space) that a batch normalization is applied, followed by a ReLU activation. Typically we would place the batch normalization *after* the activation, but once again, we will stick as close to the original GoogLeNet implementation as possible.

We start off with a 7×7 convolution with a stride of 2×2 where we are learning a total of 64 filters. A max pooling operation is immediately appealed with a kernel size of 3×3 and a stride of

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj
convolution	7×7/2	112×112×64	1						
max pool	3×3/2	56×56×64	0						
convolution	3×3/1	56×56×192	2		64	192			
max pool	3×3/2	28×28×192	0						
inception (3a)		28×28×256	2	64	96	128	16	32	32
inception (3b)		28×28×480	2	128	128	192	32	96	64
max pool	3×3/2	14×14×480	0						
inception (4a)		14×14×512	2	192	96	208	16	48	64
inception (4b)		14×14×512	2	160	112	224	24	64	64
inception (4c)		14×14×512	2	128	128	256	24	64	64
inception (4d)		14×14×528	2	112	144	288	32	64	64
inception (4e)		14×14×832	2	256	160	320	32	128	128
max pool	3×3/2	7×7×832	0						
inception (5a)		7×7×832	2	256	160	320	32	128	128
inception (5b)		7×7×1024	2	384	192	384	48	128	128
avg pool	7×7/1	1×1×1024	0						
dropout (40%)		1×1×1024	0						
linear		1×1×1000	1						
softmax		1×1×1000	0						

Figure 8.2: The full GoogLeNet architecture proposed by Szegedy et al. We'll be implementing this exact architecture and trying to replicate their accuracy.

2×2. These `CONV` and `POOL` layers reduce our input volume size down from $224 \times 224 \times 3$ all the way down to $56 \times 56 \times 64$ (notice how dramatically the spatial dimensions of the image dropped).

From there, another `CONV` layer is applied, where we learn 192 3×3 filters. A `POOL` layer then follows, reducing our spatial dimensions to $28 \times 28 \times 192$. Next, we stack two Inception modules (named 3a and 3b), followed by another `POOL`.

To learn deeper, richer features, we then stack five Inception modules (4a-4e), again followed by a `POOL`. Two more Inception modules are applied (5a and 5b), leaving us with an output volume size of $7 \times 7 \times 1024$. To avoid the usage of fully-connected layers, we can instead use *global average pooling* where we average the 7×7 volume down to $1 \times 1 \times 1024$. Dropout can then be applied to reduce overfitting. Szegedy et al. recommended using a dropout rate of 40%; however, 50% tends to be the standard for most networks – in this case, we'll follow the original implementation as close as possible and use 40%. After the dropout, we apply the only fully-connected layer in the entire architecture where the number of nodes is the total number of class labels (1,000) followed by a softmax classifier.

8.1.3 Implementing GoogLeNet

Using the table detailed above, we can now implement GoogLeNet using Python and the mxnet library. Create a new file named `mxgooglenet.py` inside the `nn.mxconv` sub-module of `pyimagesearch`, that way we can keep our Keras CNN implementations separate from our mxnet CNN implementations (something I highly recommend):

```
--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- nn
|    |    |--- __init__.py
```

```
|    |    |--- conv
|    |    |--- mxconv
|    |    |    |--- __init__.py
|    |    |    |--- mxalexnet.py
|    |    |    |--- mxgooglenet.py
|    |    |    |--- mxvggnet.py
|    |--- preprocessing
|    |--- utils
```

From there, open up the file and we'll start implementing `GoogLeNet`:

```
1  # import the necessary packages
2  import mxnet as mx
3
4  class MxGoogLeNet:
5      @staticmethod
6      def conv_module(data, K, kX, kY, pad=(0, 0), stride=(1, 1)):
7          # define the CONV => BN => RELU pattern
8          conv = mx.sym.Convolution(data=data, kernel=(kX, kY),
9              num_filter=K, pad=pad, stride=stride)
10         bn = mx.sym.BatchNorm(data=conv)
11         act = mx.sym.Activation(data=bn, act_type="relu")
12
13         # return the block
14         return act
```

On **Line 6** we define our `conv_module` method, a convenience function used to apply a layer sequence of `CONV => BN => RELU`. This function accepts an input `data` (i.e., the input from the previous layer), the kernel size `K`, the size of the kernel `kX` and `kY`, the amount of zero-padding `pad`, and finally the `stride` of the convolution. **Lines 8-11** use these inputs to build our `CONV => BN => RELU`, which we then return to the calling function on **Line 14**.

The reason we define the `conv_module` is simply for convenience – anytime we need to apply the `CONV => BN => RELU` pattern (which will be a lot), we just make a call to `conv_module`. Furthermore, doing so helps clean up our code as we won't have to *explicitly* write out every `Convolution`, `BatchNorm`, and `Activation`.

Next, we can define our `inception_module` which is identical to our implementation in Chapter 11 of the *Practitioner Bundle*, only we are now using mxnet as the implementing library rather than Keras:

```
16     @staticmethod
17     def inception_module(data, num1x1, num3x3Reduce, num3x3,
18         num5x5Reduce, num5x5, num1x1Proj):
19         # the first branch of the Inception module consists of 1x1
20         # convolutions
21         conv_1x1 = MxGoogLeNet.conv_module(data, num1x1, 1, 1)
```

On **Line 21** we create the first branch of the Inception module, which applies a total of `num1x1` 1×1 convolutions. The second branch in the Inception module is a set of 1×1 convolutions followed by 3×3 convolutions:

```
23            # the second branch of the Inception module is a set of 1x1
24            # convolutions followed by 3x3 convolutions
25            conv_r3x3 = MxGoogLeNet.conv_module(data, num3x3Reduce, 1, 1)
26            conv_3x3 = MxGoogLeNet.conv_module(conv_r3x3, num3x3, 3, 3,
27                 pad=(1, 1))
```

The exact same process is applied for the third branch, only this time we are learning 5×5 convolutions:

```
29            # the third branch of the Inception module is a set of 1x1
30            # convolutions followed by 5x5 convolutions
31            conv_r5x5 = MxGoogLeNet.conv_module(data, num5x5Reduce, 1, 1)
32            conv_5x5 = MxGoogLeNet.conv_module(conv_r5x5, num5x5, 5, 5,
33                 pad=(2, 2))
```

The final branch in our Inception module is the pool projection, which is simply a max pooling followed by 1×1 `CONV` layer:

```
35            # the final branch of the Inception module is the POOL +
36            # projection layer set
37            pool = mx.sym.Pooling(data=data, pool_type="max", pad=(1, 1),
38                 kernel=(3, 3), stride=(1, 1))
39            conv_proj = MxGoogLeNet.conv_module(pool, num1x1Proj, 1, 1)
```

The output of these four channels (i.e., the convolutions) are then concatenated along the channel dimension, forming the output of the Inception module:

```
41            # concatenate the filters across the channel dimension
42            concat = mx.sym.Concat(*[conv_1x1, conv_3x3, conv_5x5,
43                 conv_proj])
44
45            # return the block
46            return concat
```

The reason we are able to perform this concatenation is because *special care* is taken to add each of the 3×3 and 5×5 convolutions such that the output volume dimensions are the same. If the output volumes shapes for each branch *did not match* then we would be unable to perform the concatenation.

Given the `conv_module` and `inception_module`, we are now ready to create the `build` method responsible or using these building blocks to construct the GoogLeNet architecture:

```
48        @staticmethod
49        def build(classes):
50            # data input
51            data = mx.sym.Variable("data")
52
53            # Block #1: CONV => POOL => CONV => CONV => POOL
54            conv1_1 = MxGoogLeNet.conv_module(data, 64, 7, 7,
55                 pad=(3, 3), stride=(2, 2))
56            pool1 = mx.sym.Pooling(data=conv1_1, pool_type="max",
```

```
57              pad=(1, 1), kernel=(3, 3), stride=(2, 2))
58          conv1_2 = MxGoogLeNet.conv_module(pool1, 64, 1, 1)
59          conv1_3 = MxGoogLeNet.conv_module(conv1_2, 192, 3, 3,
60              pad=(1, 1))
61          pool2 = mx.sym.Pooling(data=conv1_3, pool_type="max",
62              pad=(1, 1), kernel=(3, 3), stride=(2, 2))
```

Following Figure 8.2 above, we start off by applying a `CONV => POOL => CONV => POOL` to reduce the spatial input dimensions from 224×224 pixels down to 28×28 pixels.

Keep in mind that we are simplicity applying a batch normalization and activation after every `CONV`. These layers have been omitted from the description of the code (1) for brevity and (2) because we have already discussed how they are applied inside the `conv_module` function.

Next we apply two Inception modules (3a and 3b) followed by a `POOL`:

```
64          # Block #3: (INCEP * 2) => POOL
65          in3a = MxGoogLeNet.inception_module(pool2, 64, 96, 128, 16,
66              32, 32)
67          in3b = MxGoogLeNet.inception_module(in3a, 128, 128, 192, 32,
68              96, 64)
69          pool3 = mx.sym.Pooling(data=in3b, pool_type="max",
70              pad=(1, 1), kernel=(3, 3), stride=(2, 2))
```

The exact parameters for each of the branches can be found in Table 8.2 above. Szegedy et al. determined these parameters through their own experimentations and it is recommended to leave the number of filters for each branch as they are. I provide a more thorough discussion on the assumptions we make regarding the number of filters for each `CONV` layer in the Inception module in Chapter 12 of the *Practitioner Bundle*.

To learn deeper, richer features that are more discriminative, we next apply five Inception modules (4a-4e), followed by another `POOL`:

```
72          # Block #4: (INCEP * 5) => POOL
73          in4a = MxGoogLeNet.inception_module(pool3, 192, 96, 208, 16,
74              48, 64)
75          in4b = MxGoogLeNet.inception_module(in4a, 160, 112, 224, 24,
76              64, 64)
77          in4c = MxGoogLeNet.inception_module(in4b, 128, 128, 256, 24,
78              64, 64)
79          in4d = MxGoogLeNet.inception_module(in4c, 112, 144, 288, 32,
80              64, 64)
81          in4e = MxGoogLeNet.inception_module(in4d, 256, 160, 320, 32,
82              128, 128)
83          pool4 = mx.sym.Pooling(data=in4e, pool_type="max",
84              pad=(1, 1), kernel=(3, 3), stride=(2, 2))
```

Notice the deeper we get in the architecture and the smaller the output volume becomes, the *more* filters we learn – this is a common pattern when creating Convolutional Neural Networks.

Another two Inception modules (5a and 5b) are used, followed by average pooling and dropout:

```
86          # Block #5: (INCEP * 2) => POOL => DROPOUT
87          in5a = MxGoogLeNet.inception_module(pool4, 256, 160, 320, 32,
88              128, 128)
89          in5b = MxGoogLeNet.inception_module(in5a, 384, 192, 384, 48,
90              128, 128)
91          pool5 = mx.sym.Pooling(data=in5b, pool_type="avg",
92              kernel=(7, 7), stride=(1, 1))
93          do = mx.sym.Dropout(data=pool5, p=0.4)
```

Examining **Lines 89 and 90** you can see that a total of $384 + 384 + 128 + 128 = 1024$ filters are outputted as a result from the concatenation operation during Inception module 5b. Thus, this result produces an output volume size of $7 \times 7 \times 1024$. To alleviate the need for fully-connected layers, we can average the 7×7 spatial dimensions down to $1 \times 1 \times 1024$, a common technique used in GoogLeNet, ResNet, and SqueezeNet. Doing so allows us to reduce the overall number of parameters in our network *substantially*.

Finally, we define a single dense layer for our total number of `classes` and apply a softmax classifier:

```
95          # softmax classifier
96          flatten = mx.sym.Flatten(data=do)
97          fc1 = mx.sym.FullyConnected(data=flatten, num_hidden=classes)
98          model = mx.sym.SoftmaxOutput(data=fc1, name="softmax")
99
100         # return the network architecture
101         return model
```

In our next section, we'll review the `train_googlenet.py` script used to actually train our implementation of GoogLeNet.

8.1.4 Training GoogLeNet

Now that we have implemented GoogLeNet in mxnet, we can move on to training it on the ImageNet dataset from scratch. Before we get started, let's take a look at the directory structure of our project:

```
--- mx_imagenet_googlenet
|     |--- config
|     |     |--- __init__.py
|     |     |--- imagnet_googlenet_config.py
|     |--- output/
|     |--- test_googlenet.py
|     |--- train_googlenet.py
```

Just as in the AlexNet and VGGNet chapters, the project structure is identical. The `train_googlenet.py` script will be responsible for training the actual network. We can then use `test_googlenet.py` script to evaluate the performance of a particular GoogLeNet epoch on the ImageNet testing set.

The `test_googlenet.py` file is actually identical to both `test_alexnet.py` and `test_vggnet.py` – I simply copied and included the file in this directory should you wish to modify it for your own needs.

Most importantly, we have the `imagenet_googlenet_config.py` file which controls our configuration for training GoogLeNet. I copied this file directly from `imagenet_alexnet_config.py`, renaming it to `imagenet_googlenet_config.py`. The ImageNet configuration file paths are identical to our previous experiments; the only exception is the `BATCH_SIZE` and `NUM_DEVICES` configuration:

```
# define the batch size and number of devices used for training
BATCH_SIZE = 128
NUM_DEVICES = 3
```

Here I indicate that batch sizes of 128 images should be passed through the network at a time – this fits comfortably on my Titan X GPU with 12GB of memory. If you have less memory on your GPU, simply decrease the batch size (by power of two) until the batch will fit along with the network. I then set `NUM_DEVICES = 3`, indicating I want to use three GPUs to train GoogLeNet. Using three GPUs, I was able to complete an epoch approximately every 1.7 hours, or 14 epochs in a single day. If you were to use a GPU, you can expect to reach six to eight epochs per day, making it totally feasible to train GoogLeNet is less than a week.

Now that our configuration has been updated, it's time to implement `train_googlenet.py`. I would suggest copying either `train_alexnet.py` or `train_vggnet.py` from earlier chapters as we'll only need to make *slight* modifications to `train_googlenet.py`. After copying the file, open it up, and we'll review the contents:

```
# import the necessary packages
from config import imagenet_googlenet_config as config
from pyimagesearch.nn.mxconv import MxGoogLeNet
import mxnet as mx
import argparse
import logging
import json
import os
```

On **Line 2** we import our `imagenet_googlenet_config` file so we can access the variables inside the configuration. We then import our `MxGoogLeNet` implementation from earlier in Section 8.1.3.

Next, let's parse our command line arguments and create a logging file so we can save the training process to it (enabling us to review the log and even plot training loss/accuracy):

```
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-c", "--checkpoints", required=True,
	help="path to output checkpoint directory")
ap.add_argument("-p", "--prefix", required=True,
	help="name of model prefix")
ap.add_argument("-s", "--start-epoch", type=int, default=0,
	help="epoch to restart training at")
args = vars(ap.parse_args())

# set the logging level and output file
logging.basicConfig(level=logging.DEBUG,
	filename="training_{}.log".format(args["start_epoch"]),
	filemode="w")
```

```
24
25 # load the RGB means for the training set, then determine the batch
26 # size
27 means = json.loads(open(config.DATASET_MEAN).read())
28 batchSize = config.BATCH_SIZE * config.NUM_DEVICES
```

These command line arguments are identical to our AlexNet and VGGNet chapters. First, we supply a `-checkpoints` switch, the path to a directory to serialize each of the model weights after every epoch. The `-prefix` serves as the model name – in this case, you'll likely want to supply a `-prefix` value of `googlenet` when entering the script into your terminal. In the case that we are restarting training from a specific epoch, we can supply a `-start-epoch`.

Lines 21-23 create a log file based on the `-start-epoch` so we can save the training results and later plot the loss/accuracy. In order to perform mean normalization, we load our RGB means from disk on **Line 27**.

We then derive the `batchSize` for the optimizer based on (1) the batch size specified in the configuration file and (2) the number of devices we'll be using to train the network. If we are using multiple devices (GPUs/CPUs/machines/etc.), then we'll need to scale the gradient update step by this `batchSize`.

Next, let's create the training data iterator:

```
30 # construct the training image iterator
31 trainIter = mx.io.ImageRecordIter(
32     path_imgrec=config.TRAIN_MX_REC,
33     data_shape=(3, 224, 224),
34     batch_size=batchSize,
35     rand_crop=True,
36     rand_mirror=True,
37     rotate=15,
38     max_shear_ratio=0.1,
39     mean_r=means["R"],
40     mean_g=means["G"],
41     mean_b=means["B"],
42     preprocess_threads=config.NUM_DEVICES * 2)
```

As well as the validation iterator:

```
44 # construct the validation image iterator
45 valIter = mx.io.ImageRecordIter(
46     path_imgrec=config.VAL_MX_REC,
47     data_shape=(3, 224, 224),
48     batch_size=batchSize,
49     mean_r=means["R"],
50     mean_g=means["G"],
51     mean_b=means["B"])
```

Just like in the *Practitioner Bundle*, we'll be using the Adam optimizer to train GoogLeNet:

```
53 # initialize the optimizer
54 opt = mx.optimizer.Adam(learning_rate=1e-3, wd=0.0002,
55     rescale_grad=1.0 / batchSize)
```

As you'll see in Section 8.3, I initially used SGD as the optimizer, but found that it returned sub-par results. Further accuracy was gained using Adam. I also included a small amount of weight decay (`wd=0.0002`) as recommended by Szegedy et al.

Next, let's construct the `checkpointsPath`, the path to the directory where the GoogLeNet weights will be serialized after every epoch:

```
57 # construct the checkpoints path, initialize the model argument and
58 # auxiliary parameters
59 checkpointsPath = os.path.sep.join([args["checkpoints"],
60      args["prefix"]])
61 argParams = None
62 auxParams = None
```

We'll be using the `ctrl + c` method to train GoogLeNet on ImageNet, so we need a method to (1) train GoogLeNet from scratch and (2) restart training from a specific epoch:

```
64 # if there is no specific model starting epoch supplied, then
65 # initialize the network
66 if args["start_epoch"] <= 0:
67      # build the LeNet architecture
68      print("[INFO] building network...")
69      model = MxGoogLeNet.build(config.NUM_CLASSES)
70
71 # otherwise, a specific checkpoint was supplied
72 else:
73      # load the checkpoint from disk
74      print("[INFO] loading epoch {}...".format(args["start_epoch"]))
75      model = mx.model.FeedForward.load(checkpointsPath,
76           args["start_epoch"])
77
78      # update the model and parameters
79      argParams = model.arg_params
80      auxParams = model.aux_params
81      model = model.symbol
```

Lines 66-69 handle instantiating the `MxGoogLeNet` architecture. Otherwise, **Lines 71-81** assume that we are loading a checkpoint from disk and restarting training from a specific epoch (likely after a hyperparameter update, such as learning rate).

Finally, we can compile our model:

```
83 # compile the model
84 model = mx.model.FeedForward(
85      ctx=[mx.gpu(0), mx.gpu(1), mx.gpu(2)],
86      symbol=model,
87      initializer=mx.initializer.Xavier(),
88      arg_params=argParams,
89      aux_params=auxParams,
90      optimizer=opt,
91      num_epoch=90,
92      begin_epoch=args["start_epoch"])
```

Here you can see that I am training GoogLeNet using three GPUs – you should modify **Line 85** based on the number of devices you have available on your machine. I'm also initializing my weights using the Xavier/Glorot method, the same method suggested by the original GoogLeNet paper. We'll allow our network to train for a maximum of 90 epochs, but as Section 8.3 will demonstrate, we won't need that many epochs.

I recommend setting `num_epochs` to a value *larger* than what you think you will need. One of the worst time wasters is monitoring an experiment, going to bed, and waking up the next morning, only to find that training stopped early and could have continued – you can always go back to a previous epoch if overfitting occurs or if validation loss/accuracy stagnates.

Next, let's create our callbacks and evaluation metrics:

```
94 # initialize the callbacks and evaluation metrics
95 batchEndCBs = [mx.callback.Speedometer(batchSize, 250)]
96 epochEndCBs = [mx.callback.do_checkpoint(checkpointsPath)]
97 metrics = [mx.metric.Accuracy(), mx.metric.TopKAccuracy(top_k=5),
98       mx.metric.CrossEntropy()]
```

And finally we can train our network:

```
100 # train the network
101 print("[INFO] training network...")
102 model.fit(
103       X=trainIter,
104       eval_data=valIter,
105       eval_metric=metrics,
106       batch_end_callback=batchEndCBs,
107       epoch_end_callback=epochEndCBs)
```

By this point reviewing the `train_*.py` files for ImageNet should start to feel quite redundant. I have purposely created these files such that they feel like a "framework". Whenever you want to explore a new architecture on ImageNet, copy one of these directories, update the configuration file, change any model imports, adjust the optimizer and initialization scheme, and then start training – deep learning can be complicated, but our code shouldn't be.

8.2 Evaluating GoogLeNet

To evaluate GoogLeNet on the testing set of ImageNet, we'll be using the `test_googlenet.py` script mentioned in our project structure above. This file is *identical* to both `test_alexnet.py` and `test_vggnet.py`, so we are going to skip reviewing the file to avoid redundancy. I've included `test_googlenet.py` in the directory structure and downloads associated with this book as a matter of completeness. Please refer to Chapter 6 on AlexNet for a thorough review of this file.

8.3 GoogLeNet Experiments

The results of ILSVRC 2014 indicate that GoogLeNet just slightly beat out VGGNet for the #1 position [26]; however, many deep learning researchers (myself included) have found it hard to reproduce these exact results [24] – the results should be at least on par with VGG. There is likely a parameter setting missing during training that myself and others are missing from the Szegedy et al. paper.

Regardless, it's still interesting to review the experiments and the thought process it uses to take GoogLeNet and make it achieve reasonable accuracy (i.e., better than AlexNet, but not as good as VGG).

8.3.1 GoogLeNet: Experiment #1

In my first GoogLeNet experiment, I used the *SGD optimizer* (unlike the Adam optimizer detailed in Section 8.1.4 above) with an initial learning rate of $1e-2$, the base learning rate recommended by the authors, as well as a momentum term of 0.9 and a L2 weight decay of 0.0002. As far as I could tell, these optimizer parameters were an exact replica of what Szegedy et al. report in their paper. I then started training using the following command:

```
$ python train_googlenet.py --checkpoints checkpoints --prefix googlenet
```

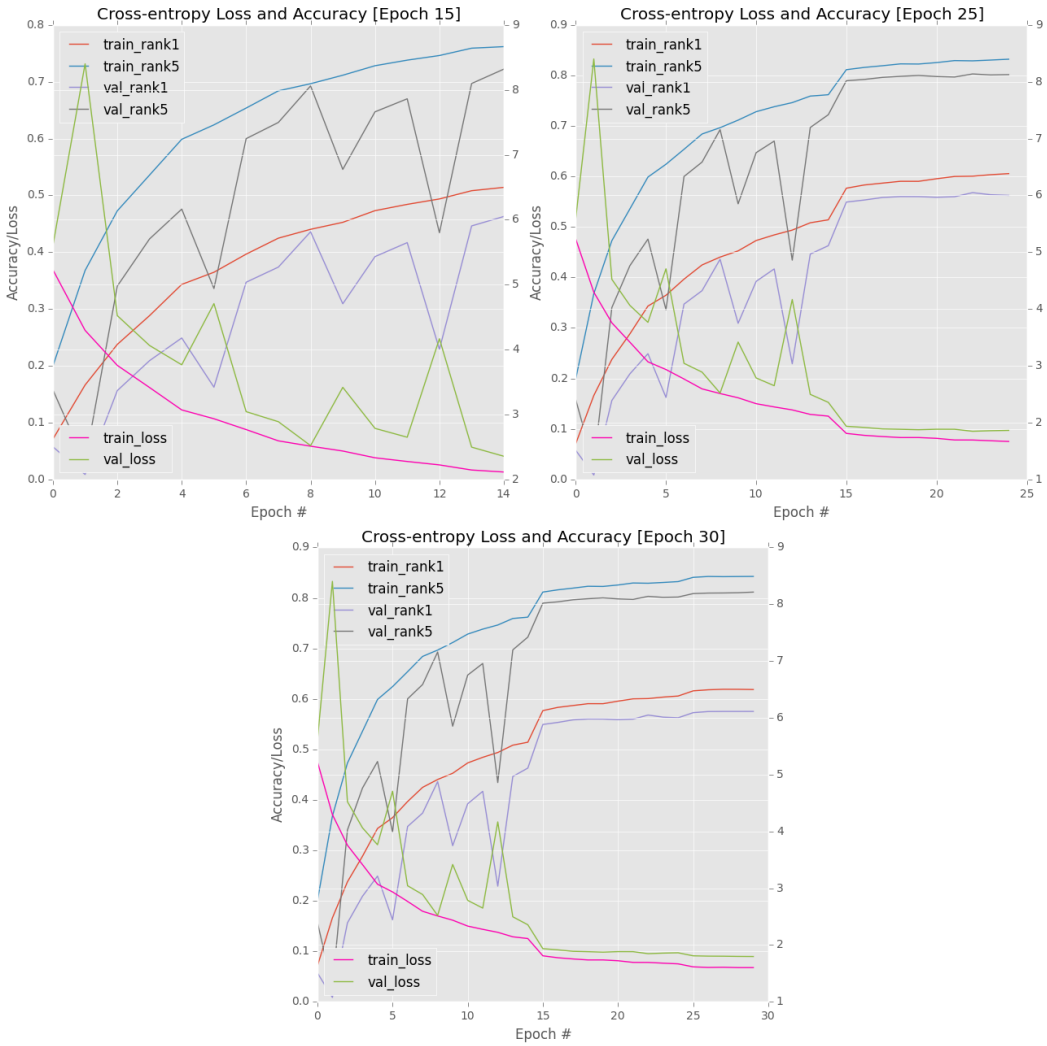


Figure 8.3: **Top-left:** The first 15 epochs of training GoogLeNet on ImageNet are highly volatile. **Top-right:** Lowering the learning rate by an order of magnitude stabilizes the volatility; however, learning quickly stagnates. **Bottom:** Lowering the learning rate to $1e-4$ does not improve results.

As my first 15 epochs demonstrate, learning is *extremely volatile* when examining the validation set (Figure 8.3, *top-left*). There are dramatic drops in validation accuracy along with significant increases in validation loss. In an effort to combat the volatility, I stopped training after the 15th epoch and lowered the learning rate to $1e-3$, then resumed training:

```
$ python train_googlenet.py --checkpoints checkpoints --prefix googlenet \
    --start-epoch 15
```

Unfortunately, this updated caused the network performance to totally stagnate after a slight bump in accuracy (Figure 8.3, *top-right*). To validate that stagnation was indeed the case, I stopped training after epoch 25, lowered the learning rate to $1e-4$, and then resumed training for another five epochs:

```
$ python train_googlenet.py --checkpoints checkpoints --prefix googlenet \
    --start-epoch 25
```

As my plot demonstrates, there is a tiny bump in accuracy as SGD is able to navigate to an area of lower loss, but overall, training has plateaued (Figure 8.3, *bottom*). After epoch 30 I terminated training and took a closer look at the results. A quick inspection of the logs revealed that I was obtaining 57.75% rank-1 and 81.15% rank-5 accuracy on the validation set. While this isn't a bad start, it's a far cry from what I expected: somewhere between AlexNet and VGGNet-level accuracies.

8.3.2 GoogLeNet: Experiment #2

Given the extreme volatility of the $1e-2$ learning rate, I decided to restart training completely, this time using a base learning rate of $1e-3$ to help smooth the learning process. I used the exact same network architecture, momentum, and L2 regularization as in the previous experiment. This approach led to steady learning; however, the process was painfully slow (Figure 8.4, *top-left*).

As learning started to slow dramatically around epoch 70, I stopped training, and lowered the learning rate to $1e-4$, then resumed training:

```
$ python train_googlenet.py --checkpoints checkpoints --prefix googlenet \
    --start-epoch 70
```

We can see a small jump in accuracy and decrease in loss as we would expect, but learning again fails to progress (Figure 8.4, *top-right*). I allowed GoogLeNet to continue to epoch 80, then stopped training once again, and lowered my learning rate by an order of magnitude to $1e-5$:

```
$ python train_googlenet.py --checkpoints checkpoints --prefix googlenet \
    --start-epoch 80
```

The result, as you might expect by now, is stagnation (Figure 8.4, *bottom*). After the 85th epoch completed, I stopped training altogether. Unfortunately, this experiment did not perform was well – I only reached 53.67% rank-1 and 77.85% rank-5 validation accuracy, worse than my first experiment.

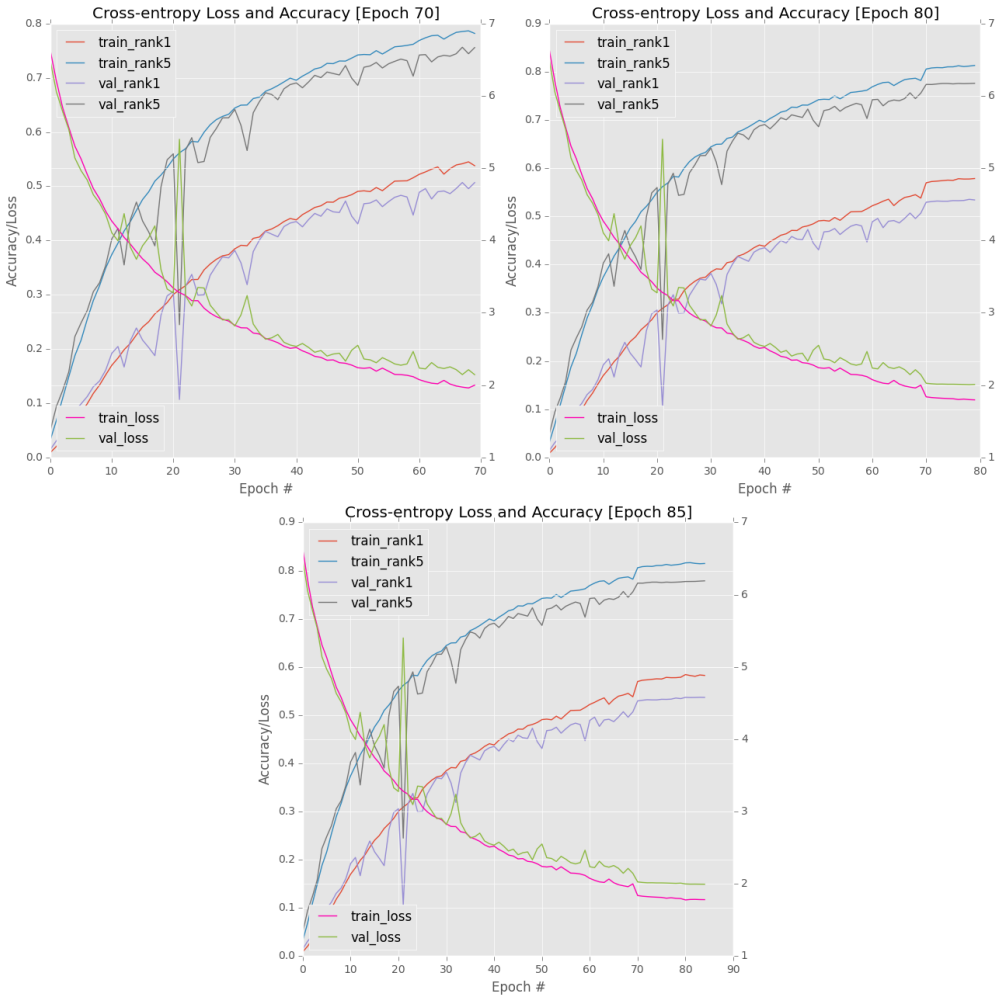


Figure 8.4: **Top-left:** The first 70 epochs demonstrate extremely slow but steady learning. **Top-right:** After learning stagnated I lowered the learning rate from $1e-3$ to $1e-4$ using the SGD optimizer. However, after the initial bump, learning quickly plateaus. **Bottom:** Further lowering of the learning rate does not improve results..

8.3.3 GoogLeNet: Experiment #3

Given my experience with GoogLeNet and Tiny ImageNet in the *Practitioner Bundle*, I decided to swap out the SGD optimizer for Adam with an initial (default) learning rate of $1e-3$. Using this approach, I then followed the learning rate decay schedule shown in Table 8.1.

The plot of accuracy and loss can be seen in Figure 8.5. As compared to other experiments, we can see that GoogLeNet + Adam quickly reached $> 50\%$ validation accuracy (less than 10 epochs, compared to the 15 epochs it took from experiment #1, and that's only because of the learning rate change). You can see my lowering of the learning rate from $1e-3$ to $1e-4$ on epoch 30 introduced a jump in accuracy. Unfortunately, there wasn't as much of a jump on epoch 45 where I switched to $1e-5$ and allowed training to continue for five more epochs.

After the 50th epoch, I stopped training entirely. Here GoogLeNet reached 63.45% rank-1 and 84.90% rank-5 accuracy on the validation set, *much* better than the previous two experiments.

I then decided to run this model on the testing set:

Epoch	Learning Rate
$1-30$	$1e-3$
$31-45$	$1e-4$
$46-50$	$1e-5$

Table 8.1: Learning rate schedule used when training GoogLeNet on ImageNet in Experiment #3.

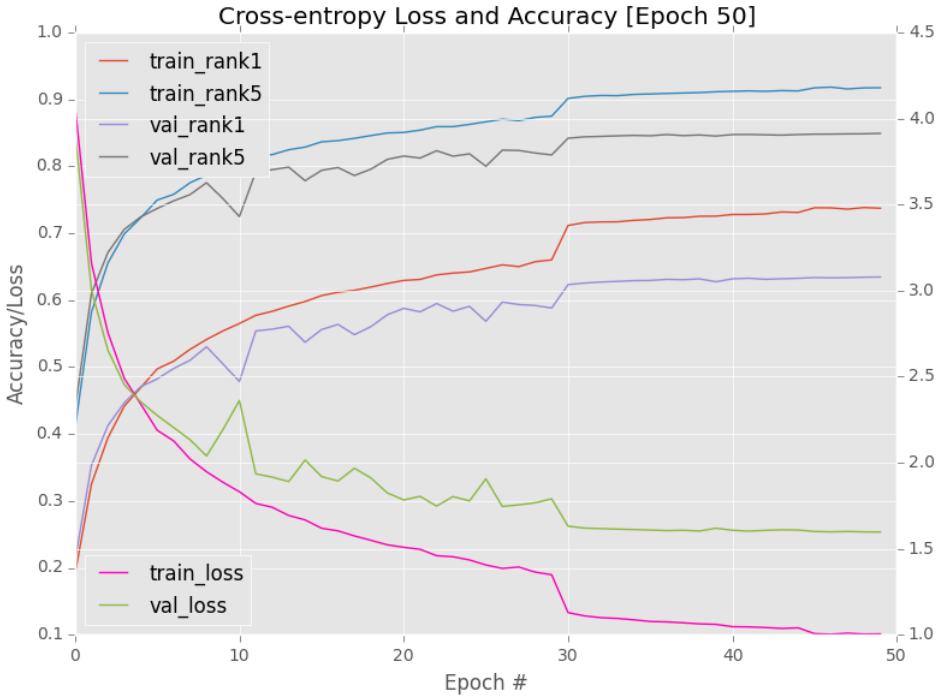


Figure 8.5: Using the Adam optimizer when training GoogLeNet yielded the best results; however, we are unable to replicate the original work by Szegedy et al. Our results are consistent with independent experiments [24] that have tried to replicate the original performance but came up slightly short.

```
$ python test_googlenet.py --checkpoints checkpoints --prefix googlenet \
    --epoch 50
[INFO] loading model...
[INFO] predicting on test data...
[INFO] rank-1: 65.87%
[INFO] rank-5: 86.48%
```

On the testing set, we obtained **65.87**% rank-1 and **87.48**% rank-5 accuracy. This result is certainly better than AlexNet, but doesn't compare to the 71.42% rank-1 and 90.03% rank-5 of VGG.

8.4 Summary

In this chapter we studied the GoogLeNet architecture proposed by Szegedy et al. in their 2014 paper, *Going Deeper with Convolutions* [23]. We then trained GoogLeNet from scratch on the ImageNet dataset, attempting to replicate their original results. However, instead of replicating the

Szegedy et al. results, we instead validated the vlfeat results [24] where our network is obtaining $\approx 65\%$ rank-1 accuracy. Based on our results, as well as the ones reported by vlfeat, it seems likely there are other extra parameters and/or training procedures we are unaware of required to obtain the VGG-level accuracy. Unfortunately, without further clarification from Szegedy et al., it's hard to pinpoint exactly *what* these extra parameters are.

We *could* continue to explore this architecture and introduce further optimizations, including:

1. Using `CONV => RELU => BN` layer ordering rather than `CONV => BN => RELU`.
2. Swapping out ReLUs for ELUs.
3. Use MSRA/He et al. initialization in combination with PreLUs.

However, it's far more interesting to study the two remaining network architectures we'll train on ImageNet – ResNet and SqueezeNet. The ResNet architecture is exciting as it introduces the residual module, capable of obtaining state-of-the-art accuracies higher than any (currently) published paper on ImageNet. We then have SqueezeNet, which obtains AlexNet-level accuracy – but does so with 50x less parameters and a 4.9MB model size.

9. Training ResNet on ImageNet

In this chapter, we'll be implementing and training the seminal ResNet architecture from scratch. ResNet is extremely important in the history of deep learning as it introduced the concept of *residual modules* and *identity mappings*. These concepts have allowed us to train networks that have > 200 layers on ImageNet and $> 1,000$ layers on CIFAR-10 – depths that were previously thought impossible to reach while successfully traiing a network.

We have reviewed the ResNet architecture in detail inside Chapter 12 of the *Practitioner Bundle*; however, this chapter will still briefly review the current incarnation of the residual module as a matter of completeness. From there, we will implement ResNet using Python and the mxnet library. Finally, we'll perform a number of experiments training ResNet from scratch ImageNet with the end goal of replicating the work by He et al. [21, 22].

9.1 Understanding ResNet

The cornerstone of ResNet is the *residual module*, first introduced by He et al. in their 2015 paper, *Deep Residual Learning for Image Recognition* [21]. The residual module consists of two branches. The first is simply a *shortcut* which connects the input to an *addition* of the second branch, a series of convolutions and activations (Figure 9.1, *left*).

However, in the same paper, it was found that *bottleneck* residual modules perform better, especially when training *deeper* networks. The bottleneck is a simple extension to the residual module. We still have our shortcut module, only now the second branch to our micro-architecture has changed. Instead of applying only two convolutions, we are now applying *three* convolutions (Figure 9.1, *middle*).

The first `CONV` consists of 1×1 filters, the second of 3×3 filters, and the third of 1×1 filters. Furthermore, the number of filters learned by the first two `CONV` is $1/4$ the number learned by the final `CONV` – this point is where the "bottleneck" comes in.

Finally, in an updated 2016 publication, *Identity Mappings in Deep Residual Networks* [22], He et al. experimented with the ordering of convolution, activation, and batch normalization layers within the residual module. They found that by applying *pre-activation* (i.e., placing the `RELU` and `BN` *before* the `CONV`) higher accuracy could be obtained (Figure 9.1, *right*).

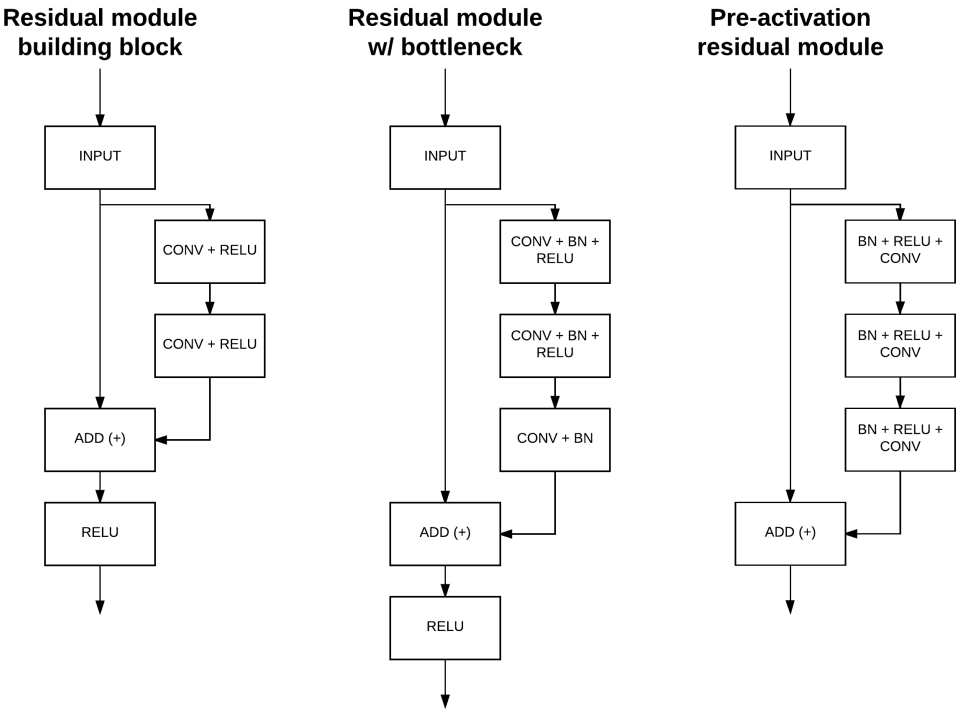


Figure 9.1: **Left:** The original residual module building block. **Center:** The residual module with a bottleneck. The bottleneck adds in an extra `CONV` layer. This module also helps reduce the dimensionality of spatial volumes. **Right:** The updated pre-activation module that changes the order of `RELU`, `CONV`, and `BN`.

We will be using the bottleneck + pre-activation version of the residual module when implementing ResNet in this chapter; however, in Section 9.5.1 I will provide results from ResNet trained *without* pre-activation to empirically demonstrate how pre-activations can boost accuracy when applied to the challenging ImageNet dataset. For more detailed information on the residual module, bottlenecks, and pre-activations, please refer to Chapter 13 of the *Practitioner Bundle* where these concepts are discussed in more detail.

9.2 Implementing ResNet

The version of ResNet we will be implementing in this chapter is ResNet50 from the He et al. 2015 publication (Figure 9.2). We call this ResNet50 because there are fifty weight layers (i.e., `CONV` or `FC`)) inside the network architecture. Given that each residual module contains three `CONV` layers, we can compute the total number of `CONV` layers inside the network via:

$$(3 \times 3) + (4 \times 3) + (6 \times 3) + (3 \times 3) = 48 \tag{9.1}$$

Finally, we add in one `CONV` layer before the residual modules followed by the `FC` layer prior to the softmax classifier, and we obtain a total of 50 layers – hence the name ResNet50.

Following Figure 9.2, our first `CONV` layer will apply a 7×7 convolution with a stride of 2×2. We'll then use a max pooling layer (the *only* max pooling layer in the entire network) with a size of 3×3 and 2×2. Doing so will enable us to reduce the spatial dimensions of the input volume from 224×224 down to 56×56 quickly.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$

Figure 9.2: Table 1 of ResNet architectures for ImageNet used by He et al. in their 2015 publication [21]. The residual modules are shown in brackets and are stacked on top of each other. The number of residual modules stacked can be determined by the $\times N$ number next to the brackets.

From there, we'll start stacking residual modules on top of each other. To start, we'll stack three residual modules (with bottleneck) to learn $K = 256$ filters (the first two `CONV` layer will learn 64 filters, respectively, while the final `CONV` in the bottleneck will learn 256 filters). Downsampling via an intermediate `CONV` layer will then occur, reducing the spatial dimensions to 28×28.

Then, four residual modules will be stacked on top of each other, each responsible for learning $K = 512$ filters. Again, an intermediate residual module will be used to reduce the volume size from 28×28 down to 14×14.

We'll again stack more residual modules (this time, six of them) where $K = 1024$. A final residual downsampling is performed to reduce spatial dimensions to 7×7, after each, we stack three residual modules to learn $K = 2048$ filters. Average 7×7 pooling is then applied (to remove the necessity of dense, fully-connected layers), followed by a softmax classifier. It's interesting to note that there is *no* dropout applied within ResNet.

I personally trained this version of ResNet50 from scratch on ImageNet using *eight* GPUs. Each epoch took approximately 1.7 hours, enabling me to complete a little over 14 epochs in a single day.

It took the authors of ResNet over *two weeks* to train the deepest variants of the architecture using 8 GPUs.

The positive here is that it only took ≈ 40 epochs to successfully train ResNet (under three days). However, it's likely that not all of us have access to eight GPUs, let alone four GPUs. If you *do* have access to four to eight GPUs, you should *absolutely* train the 50 and 101 layer variants of ResNet from scratch.

However, if you only have two to three GPUs, I would recommend training more shallow variations of ResNet by reducing the number of `stages` in the network. If you only have *one* GPU, then consider training ResNet18 or ResNet 34 (Figure 9.2). While this variant of ResNet won't obtain as high an accuracy as its deeper brothers and sisters, it will still give you practice training this seminal architecture. Again, keep in mind that ResNet and VGGNet are the two most computationally expensive networks covered in this book. All other networks, including AlexNet, GoogLeNet, and SqueezeNet can be trained with fewer GPUs.

Now that we have reviewed the ResNet architecture, let's go ahead and implement. Create a new file named `mxresnet.py` inside the `mxconv` sub-module of `pyimagesearch.nn.conv`:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- nn
|    |    |--- __init__.py
|    |    |--- conv
|    |    |--- mxconv
|    |    |    |--- __init__.py
|    |    |    |--- mxalexnet.py
|    |    |    |--- mxgooglenet.py
|    |    |    |--- mxresnet.py
|    |    |    |--- mxvggnet.py
|    |--- preprocessing
|    |--- utils
```

This is where our implementation of the ResNet architecture will live. Open up `mxresnet.py` and let's get to work:

```
1  # import the necessary packages
2  import mxnet as mx
3
4  class MxResNet:
5       # uses "bottleneck" module with pre-activation (He et al. 2016)
6       @staticmethod
7       def residual_module(data, K, stride, red=False, bnEps=2e-5,
8            bnMom=0.9):
```

On **Lines 7 and 8** we define the `residual_module` function which is very similar to our implementation from Chapter 13 of the *Practitioner Bundle*. In fact, our implementation of ResNet closely follows that of He et al. [27] and Wei Wu [28] – the parameters they proposed will be used inside this implementation. The `residual_module` accepts the following parameters:

- `data`: The input layer to the residual module.
- `K`: The number of filters the final `CONV` layer will learn. The first two `CONV` layers in the bottleneck will be learning `K / 4` layers.
- `stride`: The stride of the convolution.
- `red`: A boolean indicating whether or not we should apply an additional residual module to reduce the spatial dimensions of the volume (this downsampling is done *in between* stages).
- `bnEps`: The epsilon value of batch normalization used to prevent division by zero errors.
- `bnMom`: The momentum of the batch normalization which serves as how much "rolling average" is kept.

Let's continue to define the `residual_module`:

```
9            # the shortcut branch of the ResNet module should be
10           # initialized as the input (identity) data
11           shortcut = data
12
13           # the first block of the ResNet module are 1x1 CONVs
14           bn1 = mx.sym.BatchNorm(data=data, fix_gamma=False,
15                eps=bnEps, momentum=bnMom)
16           act1 = mx.sym.Activation(data=bn1, act_type="relu")
17           conv1 = mx.sym.Convolution(data=act1, pad=(0, 0),
```

```
18                 kernel=(1, 1), stride=(1, 1), num_filter=int(K * 0.25),
19                 no_bias=True)
```

On **Line 11** we initialize the `shortcut` branch as the input `data` to the module. This `shortcut` will later be added to the output of our bottleneck branch.

From there, **Lines 14-19** define the `BN => RELU => CONV` pattern of the first 1×1 block in the bottleneck. We call this the "pre-activation" version of ResNet since we are applying the batch normalization and activation *before* the convolution. We also leave out the bias from the convolution (`no_bias=True`) as the biases are included in the batch normalization [29].

Let's move on to the second `CONV` in the bottleneck which is responsible for learning 3×3 filters:

```
21             # the second block of the ResNet module are 3x3 CONVs
22             bn2 = mx.sym.BatchNorm(data=conv1, fix_gamma=False,
23                  eps=bnEps, momentum=bnMom)
24             act2 = mx.sym.Activation(data=bn2, act_type="relu")
25             conv2 = mx.sym.Convolution(data=act2, pad=(1, 1),
26                  kernel=(3, 3), stride=stride, num_filter=int(K * 0.25),
27                  no_bias=True)
```

The final `CONV` block in the bottleneck residual module consists of K, 1×1 filters:

```
29             # the third block of the ResNet module is another set of 1x1
30             # CONVs
31             bn3 = mx.sym.BatchNorm(data=conv2, fix_gamma=False,
32                  eps=bnEps, momentum=bnMom)
33             act3 = mx.sym.Activation(data=bn3, act_type="relu")
34             conv3 = mx.sym.Convolution(data=act3, pad=(0, 0),
35                  kernel=(1, 1), stride=(1, 1), num_filter=K, no_bias=True)
```

In the case that we are to reduce the spatial dimensions of the volume, such as when we are in between stages (`red=True`), we apply a final `CONV` layer with a stride > 1:

```
37             # if we are to reduce the spatial size, apply a CONV layer
38             # to the shortcut
39             if red:
40                  shortcut = mx.sym.Convolution(data=act1, pad=(0, 0),
41                       kernel=(1, 1), stride=stride, num_filter=K,
42                       no_bias=True)
43
44             # add together the shortcut and the final CONV
45             add = conv3 + shortcut
46
47             # return the addition as the output of the ResNet module
48             return add
```

This check allows us to apply the insights from Springenberg et al. [25] where we use convolutions with larger strides (rather than max pooling) to reduce volume size. Notice how in this case we apply the `CONV` to the output of `act1`, the first activation in the network. The reason we perform the `CONV` on the `act1` rather than the raw `shortcut` is because (1) the `data` has not been batch normalized and (2) the input `data` is the output of an addition operation. In practice, this

process tends to increase our network accuracy. Finally, we finish the residual module by adding the shortcut to the output of the bottleneck (**Line 45**) and returning the output on **Line 48**.

Let's continue on to the `build` method:

```
50      @staticmethod
51      def build(classes, stages, filters, bnEps=2e-5, bnMom=0.9):
52          # data input
53          data = mx.sym.Variable("data")
```

The `build` method requires three parameters, followed by two optional ones. The first argument is `classes`, the number of total class labels we wish our network to learn. The `stages` parameter is a list of integers, indicating the number of residual modules that will be stacked on top of each other. The `filters` list is also a list of integers, only this list contains the number of filters each `CONV` layer will learn based on what stage the residual module belongs to. Finally, we can supply optional values for the epsilon and momentum of batch normalization – we'll leave these at their default values per the recommendation of He et al. and Wei Wu.

Our next code block handles creating a series of `BN => CONV => RELU => BN => RELU => POOL` layers, the first set of layers in the network prior to stacking the residual modules on top of each other:

```
55          # Block #1: BN => CONV => ACT => POOL, then initialize the
56          # "body" of the network
57          bn1_1 = mx.sym.BatchNorm(data=data, fix_gamma=True,
58              eps=bnEps, momentum=bnMom)
59          conv1_1 = mx.sym.Convolution(data=bn1_1, pad=(3, 3),
60              kernel=(7, 7), stride=(2, 2), num_filter=filters[0],
61              no_bias=True)
62          bn1_2 = mx.sym.BatchNorm(data=conv1_1, fix_gamma=False,
63              eps=bnEps, momentum=bnMom)
64          act1_2 = mx.sym.Activation(data=bn1_2, act_type="relu")
65          pool1 = mx.sym.Pooling(data=act1_2, pool_type="max",
66              pad=(1, 1), kernel=(3, 3), stride=(2, 2))
67          body = pool1
```

Lines 57 and 58 first apply a `BN` to the input data. The batch normalization is recommended by He et al. and serves as another form of normalization (on top of mean subtraction). From there we learn a total of `filters[0]` filters, each of which are 7×7. We use a stride of 2×2 to reduce the spatial dimensions of our input volume. Batch normalization is then applied to the output of the `CONV` (**Lines 62 and 63**) followed by a ReLU activation (**Line 64**).

The first and only max pooling layer is then used on **Lines 65 and 66**. The purpose of this layer is to quickly reduce the spatial dimensions from 112×112 (after the first `CONV` layer) down to 56×56. From here, we can start stacking residual modules on top of each other (where "ResNet" gets it name).

To stack residual modules on top of each other, we need to start looping over the number of `stages`:

```
69          # loop over the number of stages
70          for i in range(0, len(stages)):
71              # initialize the stride, then apply a residual module
72              # used to reduce the spatial size of the input volume
```

```
73                  stride = (1, 1) if i == 0 else (2, 2)
74                  body = MxResNet.residual_module(body, filters[i + 1],
75                       stride, red=True, bnEps=bnEps, bnMom=bnMom)
76
77                  # loop over the number of layers in the stage
78                  for j in range(0, stages[i] - 1):
79                       # apply a ResNet module
80                       body = MxResNet.residual_module(body, filters[i + 1],
81                            (1, 1), bnEps=bnEps, bnMom=bnMom)
```

Our first `residual_module` (**Lines 74 and 75**) is designed to reduce the spatial dimensions of our input volume by 50% (when `stride=(2, 2)`). However, since this is our *first* stage of residual modules, we'll instead set the `stride=(1, 1)` to ensure the spatial dimension is *not* reduced. From there, **Lines 78-81** handle stacking residual modules.

Keep in mind that `stages` is simply a list of integers. When we implement our `train_resnet.py` script, `stages` will be set to `(3, 4, 6, 3)`, which implies that the first stage in ResNet will consist of three residual modules stacked on top of each other. Once these three residual modules have been stacked, we go back to **Lines 73-75** where we reduce spatial dimensions. The next execution of **Lines 78-81** will stack four residual modules. Once again, we then reduce spatial dimensions on **Lines 73-75**. This process repeats when stacking six residual modules and three residual modules, respectively. Luckily, by using both the `for` loops on **Line 70** and **Line 78**, we can reduce the amount of code we actually have to write (unlike in previous architectures where each layer was "hardcoded").

It's also worth noting that this stacking of the `filters` list is what makes it *very easy* for us to control the depth of a given ResNet implementation. If you want your version of ResNet to be *deeper*, simply increase the number of residual modules stacked on top of each other at each stage. In the case of ResNet152, we could set `stages=(3, 8, 36, 3)` to create a *very* deep network.

If you would rather your ResNet be more *shallow* (in the case that you want to train it using a small number of GPUs), you would *decrease* the number of residual modules stacked at each stage. For ResNet18, we may set `stages=(2, 2, 2, 2)` to create a shallow, but faster to train network.

After we are done stacking residual modules we can apply a batch normalization, activation, and then average pooling:

```
83          # apply BN => ACT => POOL
84          bn2_1 = mx.sym.BatchNorm(data=body, fix_gamma=False,
85               eps=bnEps, momentum=bnMom)
86          act2_1 = mx.sym.Activation(data=bn2_1, act_type="relu")
87          pool2 = mx.sym.Pooling(data=act2_1, pool_type="avg",
88               global_pool=True, kernel=(7, 7))
```

Keep in mind that after our final set of residual modules, our output volume has the spatial dimensions `7 x 7 x classes`. Therefore, if we apply average pooling over the 7×7 region, we will be left with an output volume of size `1 x 1 x classes` – reducing the output volume in such a way is also called *global averaging pooling*.

The last step is to simply construct a single `FC` layer with our number of `classes` and apply a softmax classifier:

```
90          # softmax classifier
91          flatten = mx.sym.Flatten(data=pool2)
92          fc1 = mx.sym.FullyConnected(data=flatten, num_hidden=classes)
```

```
93            model = mx.sym.SoftmaxOutput(data=fc1, name="softmax")
94
95            # return the network architecture
96            return model
```

9.3 Training ResNet

Now that we have implemented ResNet, we can train it on the ImageNet dataset. But first, let's define our directory structure:

```
--- mx_imagenet_resnet
|     |--- config
|     |     |--- __init__.py
|     |     |--- imagnet_resnet_config.py
|     |--- output/
|     |--- test_resnet.py
|     |--- train_resnet.py
```

I copied this directory structure from our AlexNet implementation in Chapter 6 and simply renamed all occurrences of "alexnet" to "resnet". The `train_resnet.py` script will be responsible for training our network. We can then use `test_resnet.py` to evaluate the performance of ResNet on the ImageNet testing set.

Finally, the `imagenet_resnet_config.py` file contains our configurations for our upcoming experiments. This file is identical to `imagenet_alexnet_config.py`, only I updated the `BATCH_SIZE` and `NUM_DEVICES`:

```
53 # define the batch size and number of devices used for training
54 BATCH_SIZE = 32
55 NUM_DEVICES = 8
```

Given the depth of ResNet50, I needed to reduce the `BATCH_SIZE` to 32 images on my Titan X 12GB GPU. On your own GPU, you may need to reduce this number to 16 or 8 if the GPU runs out of memory. I also set `NUM_DEVICES` to eight in order to train the network faster.

Again, I wouldn't recommend training ResNet50 on a machine without 4-8 GPUs. If you wish to train ResNet on fewer GPUs, consider using the 18 or 34 layer variant as this will be *much* faster to train. Besides VGGNet, this is the *only* network architecture I don't recommend training without multiple GPUs – and even with that said, ResNet10 can still be trained with one GPU if you are patient (and you'll still outperform AlexNet from Chapter 6).

Now that our configuration file has been updated, let's also update the `train_resnet.py` script. Just as in all previous ImageNet chapters, the `train_*.py` scripts are meant to serve as a framework, requiring us to make as little changes as possible. I will quickly review this file as we have already reviewed many times in this book. For a more detailed review of the training scripts, please refer to Chapter 6. Now, go ahead and open up `train_resnet.py` and we'll get to work:

```
1 # import the necessary packages
2 from config import imagenet_resnet_config as config
3 from pyimagesearch.nn.mxconv import MxResNet
4 import mxnet as mx
5 import argparse
6 import logging
```

```
7  import json
8  import os
```

Here we are simply importing our required Python packages. Notice how we are importing our ResNet configuration on **Line 2** along with our implementation of `MxResNet` on **Line 3**.

Next, we can parse our command line arguments and dynamically set our log file based on the `--start-epoch`:

```
10 # construct the argument parse and parse the arguments
11 ap = argparse.ArgumentParser()
12 ap.add_argument("-c", "--checkpoints", required=True,
13      help="path to output checkpoint directory")
14 ap.add_argument("-p", "--prefix", required=True,
15      help="name of model prefix")
16 ap.add_argument("-s", "--start-epoch", type=int, default=0,
17      help="epoch to restart training at")
18 args = vars(ap.parse_args())
19
20 # set the logging level and output file
21 logging.basicConfig(level=logging.DEBUG,
22      filename="training_{}.log".format(args["start_epoch"]),
23      filemode="w")
```

The `--checkpoints` switch should point to the output directory where we wish to store the serialized weights for ResNet after every epoch. The `--prefix` controls the name of the architecture, which in this case will be `resnet`. Finally, the `--start-epoch` switch is used to indicate which epoch to restart training from.

From there, we can load our RGB means from the training set and compute the `batchSize` based on the `BATCH_SIZE` and `NUM_DEVICES` in our configuration file:

```
25 # load the RGB means for the training set, then determine the batch
26 # size
27 means = json.loads(open(config.DATASET_MEAN).read())
28 batchSize = config.BATCH_SIZE * config.NUM_DEVICES
```

Let's go ahead and create our training data iterator:

```
30 # construct the training image iterator
31 trainIter = mx.io.ImageRecordIter(
32      path_imgrec=config.TRAIN_MX_REC,
33      data_shape=(3, 224, 224),
34      batch_size=batchSize,
35      rand_crop=True,
36      rand_mirror=True,
37      rotate=15,
38      max_shear_ratio=0.1,
39      mean_r=means["R"],
40      mean_g=means["G"],
41      mean_b=means["B"],
42      preprocess_threads=config.NUM_DEVICES * 2)
```

Along with the validation iterator:

```
# construct the validation image iterator
valIter = mx.io.ImageRecordIter(
     path_imgrec=config.VAL_MX_REC,
     data_shape=(3, 224, 224),
     batch_size=batchSize,
     mean_r=means["R"],
     mean_g=means["G"],
     mean_b=means["B"])
```

We'll be using the SGD optimizer to train ResNet:

```
# initialize the optimizer
opt = mx.optimizer.SGD(learning_rate=1e-1, momentum=0.9, wd=0.0001,
     rescale_grad=1.0 / batchSize)
```

We'll start with an initial learning rate of $1e-1$ and then lower it by an order of magnitude when loss/accuracy plateaus or we become concerned with overfitting. We'll also supply a momentum term of 0.9 and L2 weight decay of 0.0001, per the recommendation of He et al.

Now that our optimizer is initialized, we can construct the `checkpointsPath`, the directory where we will store the serialized weights after every epoch:

```
# construct the checkpoints path, initialize the model argument and
# auxiliary parameters
checkpointsPath = os.path.sep.join([args["checkpoints"],
     args["prefix"]])
argParams = None
auxParams = None
```

In the case that we are training ResNet from scratch, we first need to instantiate our `model`:

```
# if there is no specific model starting epoch supplied, then
# initialize the network
if args["start_epoch"] <= 0:
     # build the LeNet architecture
     print("[INFO] building network...")
     model = MxResNet.build(config.NUM_CLASSES, (3, 4, 6, 3),
          (64, 256, 512, 1024, 2048))
```

Here we will be training ResNet50 using a `stages` list of `(3, 4, 6, 3)` and `filters` list of `(64, 256, 512, 1024, 2048)`. The first `CONV` layer in ResNet (*before* any residual module) will learn $K = 64$ filters. Then, the first set of three residual modules will learn $K = 256$ filters. The number of filters learned will increase to $K = 512$ for the four residual modules stacked together. In the third stage, six residual modules will be stacked, each responsible for learning $K = 1024$ filters. And finally, the fourth stage will stack three residual modules that will learn $K = 2048$ filters.

Again, we call this architecture "ResNet50" since there are $1 + (3 \times 3) + (4 \times 3) + (6 \times 3) + (3 \times 3) + 1 = 50$ trainable layers in the network.

If we are instead restarting training from a specific epoch (presumably after adjusting the learning rate of our optimizer), we can simply load the serialized weights from disk:

```
72 # otherwise, a specific checkpoint was supplied
73 else:
74     # load the checkpoint from disk
75     print("[INFO] loading epoch {}...".format(args["start_epoch"]))
76     model = mx.model.FeedForward.load(checkpointsPath,
77         args["start_epoch"])
78
79     # update the model and parameters
80     argParams = model.arg_params
81     auxParams = model.aux_params
82     model = model.symbol
```

We are now ready to compile our model:

```
84 # compile the model
85 model = mx.model.FeedForward(
86     ctx=[mx.gpu(i) for i in range(0, config.NUM_DEVICES)],
87     symbol=model,
88     initializer=mx.initializer.MSRAPrelu(),
89     arg_params=argParams,
90     aux_params=auxParams,
91     optimizer=opt,
92     num_epoch=100,
93     begin_epoch=args["start_epoch"])
```

I'll be using eight GPUs to train ResNet (**Line 86**), but you'll want to adjust this `ctx` list based on the number of CPUs/GPUs on your machine (and also make sure to update your `NUM_DEVICES` variable in `imagenet_resnet_config.py`). Since we are training a very deep neural network, we'll want to use MSRA/He et al. initialization (**Line 88**). We'll set the `num_epoch` to train to a larger number (100), although in reality we likely won't be training for that long.

From there, we can construct our set of callbacks to monitor network performance and serialize checkpoints to disk:

```
95 # initialize the callbacks and evaluation metrics
96 batchEndCBs = [mx.callback.Speedometer(batchSize, 250)]
97 epochEndCBs = [mx.callback.do_checkpoint(checkpointsPath)]
98 metrics = [mx.metric.Accuracy(), mx.metric.TopKAccuracy(top_k=5),
99     mx.metric.CrossEntropy()]
```

The last code block handles actually training the network:

```
101 # train the network
102 print("[INFO] training network...")
103 model.fit(
104     X=trainIter,
105     eval_data=valIter,
106     eval_metric=metrics,
107     batch_end_callback=batchEndCBs,
108     epoch_end_callback=epochEndCBs)
```

9.4 Evaluating ResNet

To evaluate ResNet, we'll be using the `test_resnet.py` script mentioned in our project structure above. This script is *identical* to all other `test_*.py` scripts used in ResNet experiments. Since these scripts are the same, I will not be reviewing the Python program line-by-line. If you would like a detailed review of this evaluation script, please refer to Chapter 6.

9.5 ResNet Experiments

In this section, I will be sharing three experiments I performed when training ResNet on the ImageNet dataset. As we'll find out, ResNet can be a bit challenging to train, as due to the number of parameters in the network, it *can* overfit. However, on the positive side (and as we'll find out), training ResNet will require *fewer* epochs than previous experiments.

9.5.1 ResNet: Experiment #1

In the first two experiments on ResNet, I wanted to demonstrate how:

1. Using the residual module *without* pre-activations leads to sub-optimal performance.
2. A smaller initial learning rate can reduce the accuracy of the network.

Therefore, in this first experiment, I implemented the residual module *without* pre-activations. I then used the SGD optimizer with an initial learning rate of $1e-2$. I kept all other parameters identical to the implementation detailed in Section 9.2 above. Training was started according to the following command:

```
$ python train_resnet.py --checkpoints checkpoints --prefix resnet
```

The first 20 epochs of training can be seen in Figure 9.3 (*top-left*). This plot demonstrates that the training loss is dropping faster than the validation loss. The training accuracy is also growing at a faster rate than validation. I stopped training after epoch 20, lowered the learning rate to $1e-3$, and then resumed training:

```
$ python train_resnet.py --checkpoints checkpoints --prefix resnet \
    --start-epoch 20
```

Training continued for another 10 epochs (Figure 9.3, *top-right*). We initially see a large jump in accuracy; however, our learning quickly plateaus. To validate, I stopped training at epoch 30, adjusted my learning rate to $1e-4$, then resumed training:

```
$ python train_resnet.py --checkpoints checkpoints --prefix resnet \
    --start-epoch 30
```

I only allowed training to continue for three epochs to validate my findings (Figure 9.3, *bottom*). Sure enough, learning had essentially stalled. However, looking at rank-1 and rank-5 accuracies, I found that we reached 60.91% and 83.19%, already outperforming AlexNet.

9.5.2 ResNet: Experiment #2

My second experiment tested using a *larger* base learning rate of $1e-1$ (rather than the $1e-2$ from the first experiment). Other than the learning rate, *no other changes* were made to the experiment. I then applied a similar learning rate schedule as that of Experiment #1 (Table 9.1).

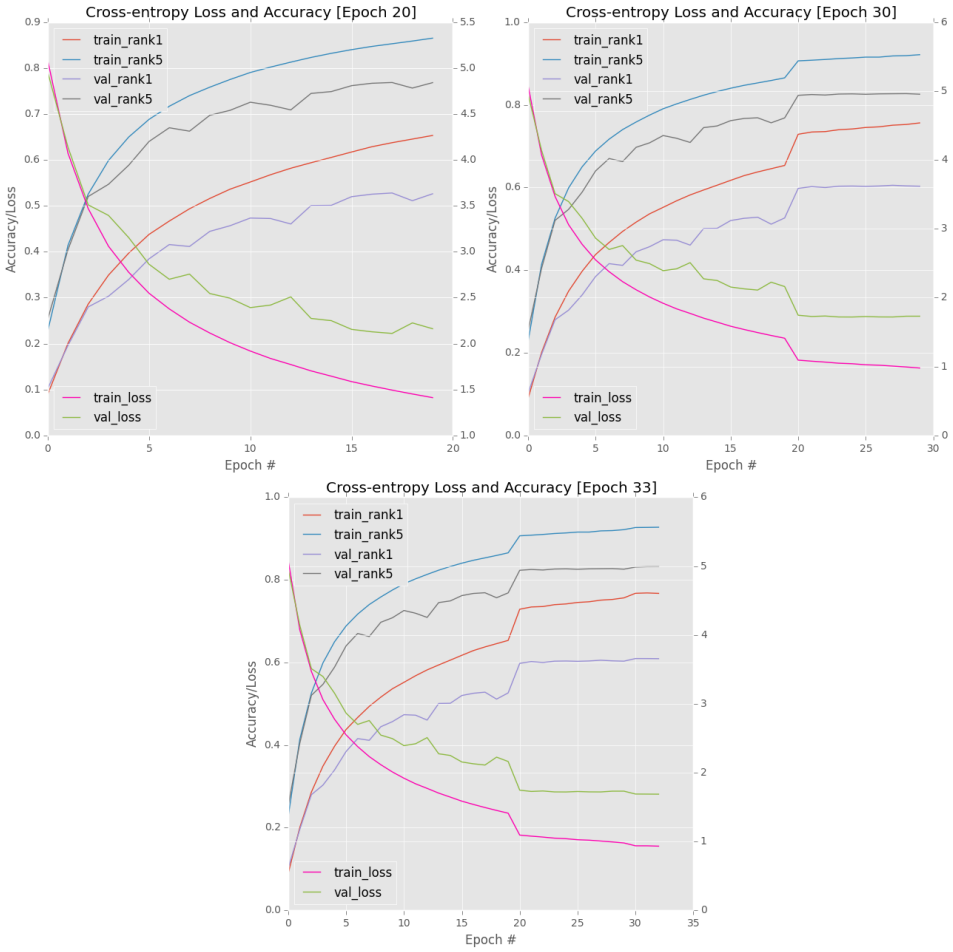


Figure 9.3: **Top-left:** First 20 epochs of training ResNet (without pre-activations) on ImageNet. **Top-right:** Lowering the learning from $1e-2$ to $1e-3$. This allows for a jump in accuracy, but learning quickly stagnates. **Bottom:** Further lowering of learning rate only marginally improves accuracy.

The results of the training process can be seen in Figure 9.4. Once again, training accuracy outpaces validation accuracy, but we are able to obtain *higher* accuracy *faster*. Furthermore, lowering the learning rate from $1e-1$ to $1e-2$ at epoch 20 gives us a *huge* jump in validation accuracy by $\approx 13\%$. A smaller jump in accuracy is obtained during the $1e-2$ to $1e-3$ drop at epoch 25; however, this small jump is accompanied by a dramatic increase in training accuracy/decrease in training loss, implying that we are at risk of modeling the training data too closely.

Once epoch 30 completed, I evaluated performance and found this version of ResNet to obtain 67.91% rank-1 and 88.13% rank-5 validation accuracy, a marked increase from our first experiment. Based on these two experiments, we can undoubtedly conclude that $1e-1$ is a better initial learning rate.

9.5.3 ResNet: Experiment #3

In my final ResNet experiment, I swapped out the original residual module for the *pre-activation* residual module (the one we implemented in Section 9.2 above). I then trained ResNet using the SGD optimizer with a base learning rate of $1e-1$, a momentum term of 0.9, and L2 weight decay

Epoch	Learning Rate
$1-20$	$1e-2$
$21-25$	$1e-3$
$26-30$	$1e-4$

Table 9.1: Learning rate schedule used when training ResNet on ImageNet in Experiment #2.

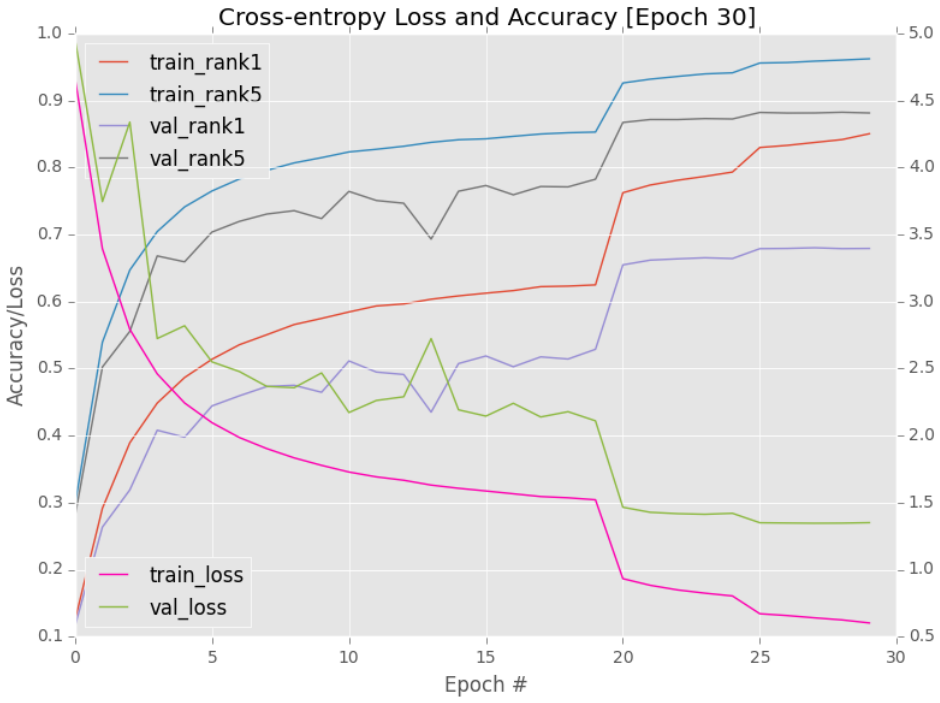


Figure 9.4: In Experiment #2 we start training ResNet with an initial learning rate of $\alpha = 1e-1$. This allows our accuracy to improvement upon Experiment #1.

of 0.0001. I started training using the following command:

```
$ python train_resnet.py --checkpoints checkpoints --prefix resnet
```

At epoch 30, we can see that training accuracy is outpacing validation accuracy, but not at an unreasonable amount (Figure 9.5, *top-left*). The ratio between the gaps at every epoch is quite consistent. However, as validation loss started to stagnate (and even rise slightly), I stopped training, lowered the learning rate from $1e-1$ to $1e-2$ and resumed training from epoch 30:

```
$ python train_resnet.py --checkpoints checkpoints --prefix resnet \
    --start-epoch 30
```

I allowed training to continue, but as my results demonstrate, we are seeing the classic signs of heavy overfitting (Figure 9.5, *top-right*). Notice how our validation loss is *increasing*, validation accuracy is *decreasing*, all while both the training accuracy and loss metrics are *improving*. Because

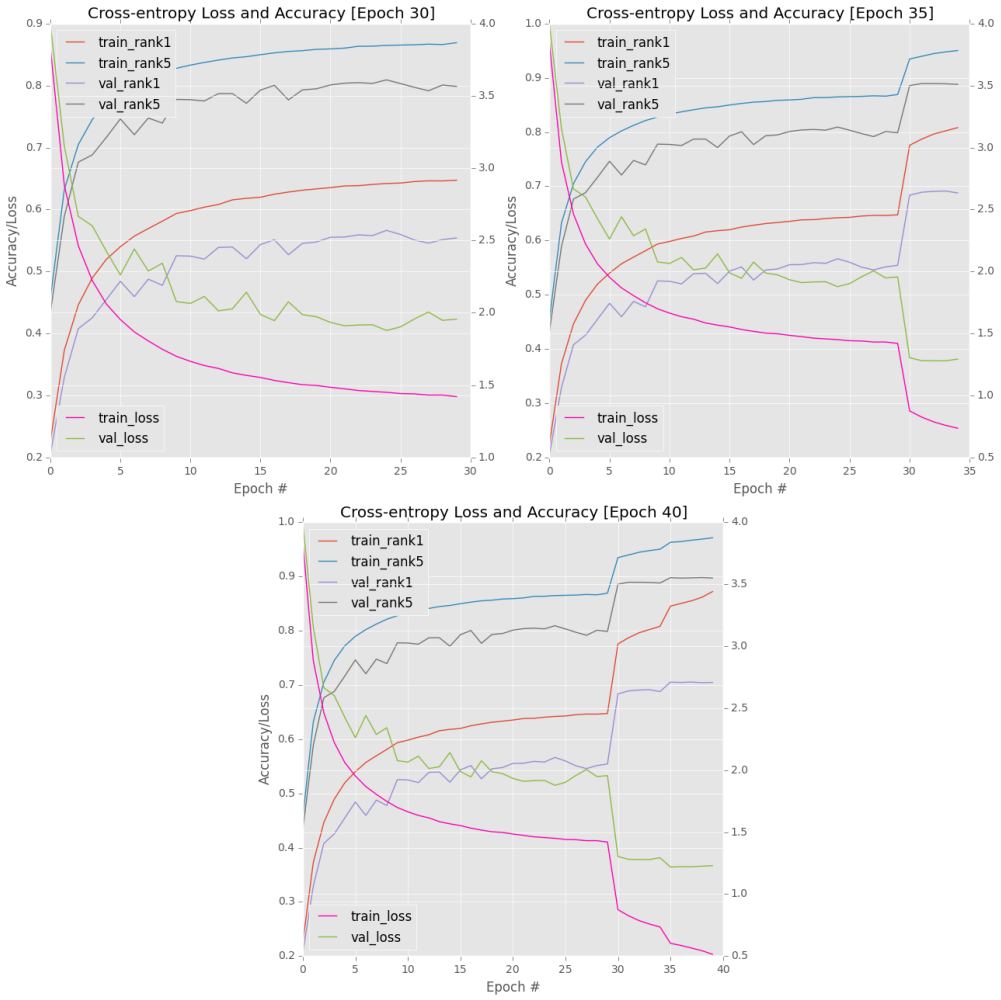


Figure 9.5: **Top-left:** Training ResNet on ImageNet using the bottleneck residual module with pre-activations. **Top-right:** Lowering the learning rate to $\alpha = 1e-2$ allows for dramatic increase in accuracy; however, we need to be careful with overfitting. **Bottom:** We can only train for another 5 epochs as overfitting is becoming problematic; however, we still obtain our best accuracy yet on ImageNet.

of this overfitting, I was forced to stop training, go back to epoch 35, adjust the learning rate from $1e-2$ to $1e-3$ and train for another five epochs:

```
$ python train_resnet.py --checkpoints checkpoints --prefix resnet \
    --start-epoch 35
```

I did not want to train for more than five epochs, because as my plots show, the training loss is decreasing at a sharply faster rate than validation loss, while training accuracy is increasing *significantly* faster than validation accuracy (Figure 9.5, *bottom*). Given that training could proceed for no longer, I stopped after epoch 40 and examined the validation metrics – 70.47% rank-1 and 89.72% rank-5, respectively. I then moved over to evaluating ResNet on the ImageNet testing set:

```
$ python test_resnet.py --checkpoints checkpoints --prefix resnet \
    --epoch 40
[INFO] loading model...
[INFO] predicting on test data...
[INFO] rank-1: 73.00%
[INFO] rank-5: 91.08%
```

Here we can see our *best results yet*, coming in at **73.01**% rank-1 and **91.08**% rank-5 on the testing set. This is *by far* our best performance on the ImageNet dataset.

However, according to the independent experiments run by vlfeat [24], we should be at approximately 75.40% rank-1 and 92.30% rank-5 accuracy. We can likely attribute our slightly lower accuracy results to the overfitting in our network. Future experiments should consider being more aggressive with regularization, including weight decay, data augmentation, and even attempting to apply dropout. In either case, we still came very close to replicating the original work by He et al. We were also able to validate that the bottleneck + pre-activation version of the residual module does lead to obtain higher accuracy than the original residual module.

9.6 Summary

In this chapter, we discussed the ResNet architecture, including both the original residual module, the bottleneck module, and the bottleneck + pre-activation module. We then trained ResNet50 from scratch on the ImageNet dataset. Overall, we were able to obtain **73.01**% rank-1 and **91.08**% rank-5 accuracy, higher than *any* of our previous experiments on the ImageNet dataset.

While ResNet is an extremely interesting architecture to study, I do not recommend training this network from scratch unless you have the time and/or financial resources to do so. ResNet50 takes *a long time* to train, hence why I needed to use eight GPUs to train the network (in order to gather the results and publish this book on time). Realistically, you could also train ResNet50 using four GPUs, but you would have to be a bit more patient. With fewer than four GPUs, I would not train ResNet50 and would encourage you to train the smaller variants, such as ResNet10 and ResNet18. While your results won't be quite as good as ResNet50, you'll still be able to learn from the process.

Keep in mind that as a deep learning practitioner, results are not everything – you should be exposing yourself to as many new projects and experiments as possible. Deep learning is part science, part art. You learn the "art" of training a deep learning network through running hundreds of experiments. Don't become discouraged if your results don't match mine or other state-of-the-art publications after your first trial – *it's all part of the process*.

Becoming a deep learning expert does not happen overnight. My goal through these series of chapters on ImageNet was to expose you to a variety of different experiments and show you how I systematically used the results from each experiment to iteratively improve upon my previous work. Keep an experiment journal of your own, noting what did (and did not) work, and you'll be able to apply this process yourself.

10. Training SqueezeNet on ImageNet

In our final chapter on training deep neural networks on the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), we will be discussing arguably my favorite deep learning architecture to date – ***SqueezeNet*** – introduced by Iandola et al. in their 2016 paper, *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and* < 0.5*MB model size* [30].

The title alone of this paper should pique your interest. Most of the previous network architectures we've discussed in this book have a model size between 100MB (ResNet) and 553MB (VGGNet). AlexNet sits in the middle of this size range too with a model weight of 249MB. The *closest* we've come to a "tiny" model size is GoogLeNet at 28MB – *but can we go smaller and still maintain state-of-the-art accuracy?*

As the work of Iandola et al. demonstrates, the answer is yes, we absolutely can decrease model size by applying a novel usage of 1×1 and 3×3 convolutions, along with *no* fully-connected layers. The end result is a model weighing in at 4.9MB, which can be further reduced to 0.5MB by *model compression*, also called *weight pruning* and *"sparsifying a model"* (setting 50% of the smallest weight values across layers to zero). In this chapter, we'll focus on the original implementation of SqueezeNet. The concept of model compression, including quantization, is outside the scope of this book but can be found in the relevant academic publication [30].

10.1 Understanding SqueezeNet

In this section we will:

1. Review the Fire module, the critical micro-architecture responsible for reducing model parameters *and* maintaining a high level of accuracy.
2. Review the *entire* SqueezeNet architecture (i.e., the macro-architecture).
3. Implement SqueezeNet by hand using mxnet.

Let's go ahead and start with a discussion on the Fire module.

10.1.1 The Fire Module

The work by Iandola et al. served two purposes. The first was to provide a case for more research into designing Convolutional Neural Networks that can be trained with a *dramatic* reduction in the

number of parameters (while still obtaining a high level of accuracy). The second contribution was the Fire module itself, the actual *implementation* of this goal.

The Fire module is quite clever in how it works, as it relies on an *expand* and *reduce* phase consisting of only 1×1 and 3×3 convolutions – a visualization of the Fire module can be seen in Figure 10.1.

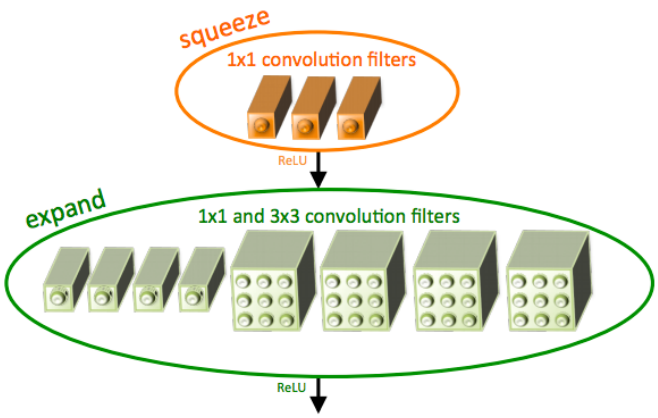


Figure 10.1: The Fire module in SqueezeNet. First a set of 1×1 filters are learned to *reduce* the dimensionality of the volume. Then a combination of 3×3 and 1×1 filters are learned and concatenated along the channel dimension to *expand* the output volume.

The *squeeze* phase of the Fire module learns a set of 1×1 filters, followed by a ReLU activation function. The number of squeeze filters learned is always *smaller* than the volume size input to the squeeze; hence, the squeeze process can be seen as a form of dimensionality reduction.

Secondly, by using 1×1 filters, we are *only* learning local features. Typically we would use a larger 3×3 or even 5×5 kernel to learn features capturing spatial information of pixels lying close together in the input volume. This is not the case for 1×1 filters – instead of trying to learn the spatial relationship amongst neighboring pixels, we are *strictly* interested in the relationship this pixel has amongst its channels.

Finally, even though we are reducing the dimensionality during the squeeze (since the number of filters learned during the squeeze stage is always smaller than the filters inputted to it), we actually have a chance to add *more nonlinearity* as the ReLU activation is applied after every 1×1 convolution.

After we apply a *squeeze*, the output is fed into an *expand*. During the expand stage we learn a combination of 1×1 and 3×3 convolutions (again, keeping in mind that the *output* of the squeeze is the *input* to the expand). The 3×3 convolutions are especially helpful here as they can allow us to capture spatial information from the original 1×1 filters.

In the SqueezeNet architecture, we learn N, 1×1 expand filters followed by N, 3×3 filters – in the Iandola et al. paper, N is $4x$ larger than the number of squeeze filters, hence why we call this the "expand" stage of the micro-architecture. First we "squeeze" the dimensionality of the input, then we "expand" it. The output of both the 1×1 and 3×3 expands are concatenated across the filter dimension, serving as the final output of the module. The 3×3 convolutions are zero-padded with one pixel in each direction to ensure the outputs can be stacked.

The Fire module is able to keep the number of total parameters small due to the dimensionality reduction during the squeeze stage. For example, suppose the input to a Fire module was $55 \times 55 \times 96$. We could then apply a squeeze with 16 filters, reducing the volume size to $55 \times 55 \times 16$. The expand could then increase the number of filters learned from 16 to 128 (64 filters for the 1×1 convolutions and 64 filters for the 3×3 convolutions). Furthermore, by using many 1×1 filters

rather than 3×3 filters, fewer parameters are required. This process allows us to reduce spatial volume size throughout the network while keeping the number of filters learned relatively low (compared to other network architectures such as AlexNet, VGGNet, etc.). We'll discuss the second huge parameter saving technique (no fully-connected layers) in the following section.

10.1.2 SqueezeNet Architecture

You can find the entire SqueezeNet architecture, including output volume size, summarized in Figure 10.2.

layer name/type	output size	filter size / stride (if not a fire layer)	depth	s_{1x1} (#1x1 squeeze)	e_{1x1} (#1x1 expand)	e_{3x3} (#3x3 expand)
input image	224x224x3					
conv1	111x111x96	7x7/2 (x96)	1			
maxpool1	55x55x96	3x3/2	0			
fire2	55x55x128		2	16	64	64
fire3	55x55x128		2	16	64	64
fire4	55x55x256		2	32	128	128
maxpool4	27x27x256	3x3/2	0			
fire5	27x27x256		2	32	128	128
fire6	27x27x384		2	48	192	192
fire7	27x27x384		2	48	192	192
fire8	27x27x512		2	64	256	256
maxpool8	13x12x512	3x3/2	0			
fire9	13x13x512		2	64	256	256
conv10	13x13x1000	1x1/1 (x1000)	1			
avgpool10	1x1x1000	13x13/1	0			

Figure 10.2: The SqueezeNet architecture, reproduced from Table 1 of Iandola et al. [30]. Given an input input we apply a `CONV` layer with a 2×2 stride followed by max pooling to quickly reduce volume size. A series of fire modules are then stacked on top of each other. The $s_{1\times1}$ column indicates the number of squeeze filters learned in the first `CONV` layer of the fire module. The $e_{1\times1}$ and $e_{3\times3}$ indicate the number of filters learned in the second set of `CONV` layers, respectively.

ReLU activations are assumed to be applied after *every* convolution and are removed from this table as their usage is implied (and to help save space including this table in the book).

It's interesting to note that in the original Iandola et al. publication they reported the input image to have spatial dimensions $224 \times 224 \times 3$; however, if we apply the following calculation used in the *Starter Bundle* (Chapter 11):

$$((224 - 7 + 2(0))/4) + 1 = 55.25 \tag{10.1}$$

Then we know that a 7×7 convolution with a stride of 2×2 cannot possibly fit. Similar to the AlexNet confusion in Chapter 6, I'm assuming this to be a typo in the original work. Using an input of $227 \times 227 \times 3$ pixels allows the convolutions to be tiled and provides the exact output volume size for all other layers as detailed in the original Iandola et al. publication. Therefore, in this book, we will assume the input image to be $227 \times 227 \times 3$.

After an input image is fed into the network, 96 filters are learned, each of size 7×7. A stride of 2×2 is used to reduce the spatial dimensions from 227×227 down to 111×111. To further reduce spatial dimensions, max pooling is applied with a pool size of 3×3 and a stride of 2×2, yielding an output volume of $55 \times 55 \times 96$.

Next, we apply three fire modules. The first two fire modules use apply 16 squeeze filters, reducing the input volume size from $55 \times 55 \times 96$ down to $55 \times 55 \times 16$. The expand then applies 64, 1×1 filters and 64 3×3 filters – these 128 filters are concatenated along the channel dimension where the output spatial size is $55 \times 55 \times 128$. The third fire module increases the number of 1×1 squeeze filters to 32, along with the number of 1×1 and 3×3 expands to 64, respectively. Prior to the max pooling operation, our spatial dimension size is $55 \times 55 \times 256$. A max pooling layer with a 3×3 pool size and 2×2 stride reduces our spatial dimensions down to $27 \times 27 \times 256$.

From there we apply four fire models sequentially. The first fire module uses 32, 1×1 squeeze filters and 128 filters for both the 1×1 and 3×3 expand, respectively.

The second two fire modules increase the number of 1×1 squeezes to 48, as well as increases the 1×1 and 3×3 expands to 192, resulting in an output volume size of $27 \times 257 \times 384$. The final fire module performs another squeeze increase, this time to 64, 1×1 filters. The number of 1×1 and 3×3 expands also increases to 256, respectfully. After concatenating these expand filters, our volume size is now $27 \times 27 \times 512$. Another max pooling operation is then applied to reduce our spatial dimensions to $13 \times 13 \times 512$.

A final fire module is applied, increasing the 1×1 squeeze filters to 64. The expand filters are then increased to 256 each for the 1×1 and 3×3, respectively. Concatenating along the channel dimension, we arrive at an output volume size of $13 \times 13 \times 512$.

One last fire module is applied again, identical to fire8 earlier in the table. Dropout is then applied directly after fire9 with a probability of 50% – this done to help reduce overfitting. The final convolution (conv10) consists of 1×1 filters with N total filters, where N should equal the total number of class labels – in the case of ImageNet, $N = 1000$.

We can perform global average pooling across the entire 13×13 volume to reduce the $13 \times 13 \times 1000$ down to a $1 \times 1 \times 1000$ volume. We take these activations and then pass them through a softmax layer to obtain our final output probabilities for each of the N class labels.

As you can see, no fully-connected layers are applied in this network architecture. Similar to GoogLeNet and ResNet, using global average pooling can help reduce (and often eliminate entirely) the need for `FC` layers. Removing the `FC` layers also has the added benefit of reducing the number of parameters require by the network *substantially*. Keep in mind that all nodes in a fully-connected layer are densely connected (i.e., every node in the current layer codes with every other node in the subsequent layer). However, convolutional layers are by definition sparse and thus require less memory. Anytime we can replace a set of fully-connected layers with convolution and average pooling we can *dramatically* reduce the number of parameters required by our network.

10.1.3 Implementing SqueezeNet

Now that we have reviewed the SqueezeNet architecture, let's go ahead and implement it. Create a new file named `mxsqueezenet.py` inside the `mxconv` sub-module of `pyimagesearch.nn.conv`:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- nn
|    |    |--- __init__.py
|    |    |--- conv
|    |    |--- mxconv
|    |    |    |--- __init__.py
```

```
|    |    |    |--- mxalexnet.py
|    |    |    |--- mxgooglenet.py
|    |    |    |--- mxresnet.py
|    |    |    |--- mxsqueezenet.py
|    |    |    |--- mxvggnet.py
|    |--- preprocessing
|    |--- utils
```

This file is where our implementation of the SqueezeNet network will live. Open up `mxsqueezenet.py` and we'll get to work:

```
1  # import the necessary packages
2  import mxnet as mx
3
4  class MxSqueezeNet:
5      @staticmethod
6      def squeeze(input, numFilter):
7          # the first part of a FIRE module consists of a number of 1x1
8          # filter squeezes on the input data followed by an activation
9          squeeze_1x1 = mx.sym.Convolution(data=input, kernel=(1, 1),
10             stride=(1, 1), num_filter=numFilter)
11         act_1x1 = mx.sym.LeakyReLU(data=squeeze_1x1,
12             act_type="elu")
13
14         # return the activation for the squeeze
15         return act_1x1
```

Line 4 defines our `MxSqueezenet` class – all methods responsible for building and creating the SqueezeNet architecture will exist inside this class. **Line 6** then defines the `squeeze` method which is part of the Fire module discussed in the previous section. The `squeeze` function requires an `input` (i.e., the preceding layer) along with `numFilter`, the number of 1×1 filters the `CONV` layer should learn (**Lines 9 and 10**).

Once we compute the 1×1 convolutions, we pass the values through an activation function on **Lines 11 and 12**. In the original Iandola et al. publication they used the standard ReLU activation function, but here we are going to use an ELU. As previous experiments with different architectures have demonstrated, replacing ReLUs with ELUs can increase classification accuracy. I will justify this claim in Section 10.4.4 below. Finally, the 1×1 activations are returned to the calling function on **Line 15**.

Now that the `squeeze` function is defined, let's also create the `fire` function:

```
17     @staticmethod
18     def fire(input, numSqueezeFilter, numExpandFilter):
19         # construct the 1x1 squeeze followed by the 1x1 expand
20         squeeze_1x1 = MxSqueezeNet.squeeze(input, numSqueezeFilter)
21         expand_1x1 = mx.sym.Convolution(data=squeeze_1x1,
22             kernel=(1, 1), stride=(1, 1), num_filter=numExpandFilter)
23         relu_expand_1x1 = mx.sym.LeakyReLU(data=expand_1x1,
24             act_type="elu")
```

The `fire` method requires three parameters:

1. `input`: The input layer to the current Fire module.

2. `numSqueezeFilter`: The number of squeeze filters to learn.
3. `numExpandFilter`: The number of expand filters to learn. As per the Iandola et al. implementation, we'll be using the same number of expand filters for both the 1×1 and 3×3 convolutions, respectively.

Line 20 makes a call to the `squeeze` function, reducing our `input` spatial dimensions. Based off of `squeeze_1x1`, we can compute the 1×1 expand on **Lines 21 and 22**. Here indicate that the input to the `CONV` layer should be the `squeeze_1x1` output and that we wish to learn `numExpandFilter`, each of size 1×1. An ELU is applied to the output of the convolution on **Lines 23 and 24**.

We perform a similar operation for the 3×3 expand below:

```
26          # construct the 3x3 expand
27          expand_3x3 = mx.sym.Convolution(data=squeeze_1x1, pad=(1, 1),
28               kernel=(3, 3), stride=(1, 1), num_filter=numExpandFilter)
29          relu_expand_3x3 = mx.sym.LeakyReLU(data=expand_3x3,
30               act_type="elu")
31
32          # the output of the FIRE module is the concatenation of the
33          # activation for the 1x1 and 3x3 expands along the channel
34          # dimension
35          output = mx.sym.Concat(relu_expand_1x1, relu_expand_3x3,
36               dim=1)
37
38          # return the output of the FIRE module
39          return output
```

This time we learn `numExpandFilter`, each of which are 3×3 using `squeeze_1x1` as an input. We apply 1×1 zero padding here to ensure the spatial dimensions are not reduced so we can concatenate along the channel dimension later in this function. An ELU is then also applied to the output of `expand_3x3`. Once we have the activations for both the 1×1 and 3×3 expands, we can concatenate them along the channel dimension on **Lines 35 and 36**. This concatenation is returned to the calling method on **Line 39**.

To help you visualize the Fire module further, please refer to Figure 10.1 above. Here you can see that the Fire module takes an input layer and applies a squeeze, consisting of a number of 1×1 kernels. This squeeze then branches into two forks. In one fork we compute the 1×1 expand, while the second fork computes the 3×3 expand. The two forks meet again and are concatenated along the channel dimension to serve as the final output of the channel module.

Let's continue our discussion on implementing SqueezeNet as we've now reached the all important `build` method:

```
41      @staticmethod
42      def build(classes):
43          # data input
44          data = mx.sym.Variable("data")
45
46          # Block #1: CONV => RELU => POOL
47          conv_1 = mx.sym.Convolution(data=data, kernel=(7, 7),
48               stride=(2, 2), num_filter=96)
49          relu_1 = mx.sym.LeakyReLU(data=conv_1, act_type="elu")
50          pool_1 = mx.sym.Pooling(data=relu_1, kernel=(3, 3),
51               stride=(2, 2), pool_type="max")
```

The `build` method accepts a single parameter, the total number of `classes` we wish to learn. In the case of ImageNet, we'll set `classes=1000`.

Line 44 then instantiates our `data` variable, the input batch to our network. Given the `data`, we define the first convolution layer in SqueezeNet. This layer learns 96 filters, each of which are 7×7 with a stride of 2×2. An ELU activation is applied (**Line 49**) before we reduce the spatial dimensions via max pooling (**Lines 50 and 51**).

Next comes our three fire modules followed by another max pooling operation:

```
53          # Block #2-4: (FIRE * 3) => POOL
54          fire_2 = MxSqueezeNet.fire(pool_1, numSqueezeFilter=16,
55              numExpandFilter=64)
56          fire_3 = MxSqueezeNet.fire(fire_2, numSqueezeFilter=16,
57              numExpandFilter=64)
58          fire_4 = MxSqueezeNet.fire(fire_3, numSqueezeFilter=32,
59               numExpandFilter=128)
60          pool_4 = mx.sym.Pooling(data=fire_4, kernel=(3, 3),
61              stride=(2, 2), pool_type="max")
```

Following along with Figure 10.2 above, we then apply our four fire modules and then another max pooling:

```
63          # Block #5-8: (FIRE * 4) => POOL
64          fire_5 = MxSqueezeNet.fire(pool_4, numSqueezeFilter=32,
65              numExpandFilter=128)
66          fire_6 = MxSqueezeNet.fire(fire_5, numSqueezeFilter=48,
67              numExpandFilter=192)
68          fire_7 = MxSqueezeNet.fire(fire_6, numSqueezeFilter=48,
69              numExpandFilter=192)
70          fire_8 = MxSqueezeNet.fire(fire_7, numSqueezeFilter=64,
71              numExpandFilter=256)
72          pool_8 = mx.sym.Pooling(data=fire_8, kernel=(3, 3),
73              stride=(2, 2), pool_type="max")
```

At this point, we are at the deepest part of the network – we need to apply another fire module, dropout to reduce overfitting, a convolution to adjust the volume size to `13 x 13 x classes`, and finally global averaging pooling to average the 13×13 spatial dimensions down to `1 x 1 x classes`:

```
75          # Block #9-10: FIRE => DROPOUT => CONV => RELU => POOL
76          fire_9 = MxSqueezeNet.fire(pool_8, numSqueezeFilter=64,
77              numExpandFilter=256)
78          do_9 = mx.sym.Dropout(data=fire_9, p=0.5)
79          conv_10 = mx.sym.Convolution(data=do_9, num_filter=classes,
80              kernel=(1, 1), stride=(1, 1))
81          relu_10 = mx.sym.LeakyReLU(data=conv_10, act_type="elu")
82          pool_10 = mx.sym.Pooling(data=relu_10, kernel=(13, 13),
83              pool_type="avg")
```

Our final code block handles flattening the output of the average pooling operation and creating the softmax output:

```
85          # softmax classifier
86          flatten = mx.sym.Flatten(data=pool_10)
87          model = mx.sym.SoftmaxOutput(data=flatten, name="softmax")
88
89          # return the network architecture
90          return model
```

As you can see, our implementation follows the network architecture detailed in Figure 10.2 above. The question is, can we train it on ImageNet? And how challenging will it be? To answer these questions, let's move on to the next section.

10.2 Training SqueezeNet

Now that we have implemented SqueezeNet, we can train it on the ImageNet dataset. But first, let's define the project structure:

```
--- mx_imagenet_squeezenet
|     |--- config
|     |     |--- __init__.py
|     |     |--- imagnet_squeezenet_config.py
|     |--- output/
|     |--- test_squeezenet.py
|     |--- train_squeezenet.py
```

The project structure is essentially identical to all other ImageNet projects earlier in this book. The `train_squeezenet.py` script will be responsible for training the actual network. We'll then use `test_squeezenet.py` to evaluate a trained SqueezeNet model on ImageNet. Finally, the `imagenet_squeezenet_config.py` file contains our configurations for the experiment.

When defining this project structure, I copied the entire `mx_imagenet_alexnet` directory and renamed the files to say `squeezenet` instead of `alexnet`. The ImageNet configuration paths are identical to all previous experiments, so let's briefly take a look at the `BATCH_SIZE` and `NUM_DEVICES` configuration:

```
53 # define the batch size and number of devices used for training
54 BATCH_SIZE = 128
55 NUM_DEVICES = 3
```

Here I indicate that a `BATCH_SIZE` of 128 should be used, which fits comfortably on my 12GB Titan X GPU. If you have a GPU with less memory, simply reduce the batch size such that the network fits on your GPU. I then used three GPUs for training as I was in a rush to gather results for this chapter. This network can also easily be trained with one or two GPUs with a small amount of patience.

Now that our configuration file has been updated, let's also update `train_squeeznet.py`. Just as in all previous ImageNet chapters, the `train_*.py` scripts are meant to serve as a framework, requiring us to make *as few changes as possible*; therefore, I will quickly review this file, highlighting the changes, and defer a thorough review of the training scripts to Chapter 6. Go ahead and open up `train_squeezenet.py` and we can get started:

```
1  # import the necessary packages
2  from config import imagenet_squeezenet_config as config
3  from pyimagesearch.nn.mxconv import MxSqueezeNet
4  import mxnet as mx
5  import argparse
6  import logging
7  import json
8  import os
```

Lines 2-8 import our required Python packages. Notice how we are importing the SqueezeNet configuration file (**Line 2**) along with the `MxSqueezeNet` class (our implementation of the SqueezeNet architecture) on **Line 3**.

Let's parse our command line arguments and create our logging file so we can log the training process to it:

```
10 # construct the argument parse and parse the arguments
11 ap = argparse.ArgumentParser()
12 ap.add_argument("-c", "--checkpoints", required=True,
13      help="path to output checkpoint directory")
14 ap.add_argument("-p", "--prefix", required=True,
15      help="name of model prefix")
16 ap.add_argument("-s", "--start-epoch", type=int, default=0,
17      help="epoch to restart training at")
18 args = vars(ap.parse_args())
19
20 # set the logging level and output file
21 logging.basicConfig(level=logging.DEBUG,
22      filename="training_{}.log".format(args["start_epoch"]),
23      filemode="w")
24
25 # load the RGB means for the training set, then determine the batch
26 # size
27 means = json.loads(open(config.DATASET_MEAN).read())
28 batchSize = config.BATCH_SIZE * config.NUM_DEVICES
```

These are the exact same command line arguments as in our previous experiments. We need to supply a `--checkpoints` directory to serialize the model weights after each epoch, a `--prefix` (i.e., name) of the model, and optionally a `--start-epoch` to resume training from. **Lines 21 and 22** dynamically create the logging file based on the starting epoch. We then load our RGB means on **Line 27** so we can apply mean subtraction normalization. The `batchSize` is derived on **Line 28** based on the total number of devices used to train SqueezeNet.

Next, let's create the training data iterator:

```
30 # construct the training image iterator
31 trainIter = mx.io.ImageRecordIter(
32      path_imgrec=config.TRAIN_MX_REC,
33      data_shape=(3, 227, 227),
34      batch_size=batchSize,
35      rand_crop=True,
36      rand_mirror=True,
37      rotate=15,
38      max_shear_ratio=0.1,
```

```
39      mean_r=means["R"],
40      mean_g=means["G"],
41      mean_b=means["B"],
42      preprocess_threads=config.NUM_DEVICES * 2)
```

As well as the validation data iterator:

```
44  # construct the validation image iterator
45  valIter = mx.io.ImageRecordIter(
46      path_imgrec=config.VAL_MX_REC,
47      data_shape=(3, 227, 227),
48      batch_size=batchSize,
49      mean_r=means["R"],
50      mean_g=means["G"],
51      mean_b=means["B"])
```

SGD will once again be used to train the network:

```
53  # initialize the optimizer
54  opt = mx.optimizer.SGD(learning_rate=1e-2, momentum=0.9, wd=0.0002,
55      rescale_grad=1.0 / batchSize)
```

Iandola et al. recommended a learning rate of $4e-2$ in their original publication; however, I found this learning rate to be far too large. Learning was *extremely volatile* and it made the network hard to converge; thus, in all of my experiments that obtained "good" accuracy, I used a $1e-2$ initial learning rate. The momentum term of 0.9 and L2 weight day of 0.0002 are those recommended by Iandola et al.

Now that our optimizer is initialized, we can construct the `checkpointsPath`, the directory where we will store the serialized weights after every epoch:

```
57  # construct the checkpoints path, initialize the model argument and
58  # auxiliary parameters
59  checkpointsPath = os.path.sep.join([args["checkpoints"],
60      args["prefix"]])
61  argParams = None
62  auxParams = None
```

Next, we can handle if we are (1) training SqueezeNet from the very first epoch or (2) restarting training from a *specific* epoch:

```
64  # if there is no specific model starting epoch supplied, then
65  # initialize the network
66  if args["start_epoch"] <= 0:
67      # build the LeNet architecture
68      print("[INFO] building network...")
69      model = MxSqueezeNet.build(config.NUM_CLASSES)
70
71  # otherwise, a specific checkpoint was supplied
72  else:
73      # load the checkpoint from disk
```

```
74      print("[INFO] loading epoch {}...".format(args["start_epoch"]))
75      model = mx.model.FeedForward.load(checkpointsPath,
76          args["start_epoch"])
77
78      # update the model and parameters
79      argParams = model.arg_params
80      auxParams = model.aux_params
81      model = model.symbol
```

Lines 66-69 handle if we are training SqueezeNet with no prior checkpoint. If this is indeed the case, we instantiate `MxSqueezeNet` on **Line 69** using the supplies number of class labels found in our configuration file (1,000 for ImageNet). Otherwise, **Lines 71-81** assume we are loading a checkpoint from disk and restarting training from a specific epoch.

Finally, we can compile our model:

```
83  # compile the model
84  model = mx.model.FeedForward(
85      ctx=[mx.gpu(0), mx.gpu(1), mx.gpu(2)],
86      symbol=model,
87      initializer=mx.initializer.Xavier(),
88      arg_params=argParams,
89      aux_params=auxParams,
90      optimizer=opt,
91      num_epoch=90,
92      begin_epoch=args["start_epoch"])
```

Here you can see that I am training SqueezeNet with three GPUs. SqueezeNet can also be trained with a single GPU (although it will take longer, of course). Feel free to speed up or slow down the training time by allocating more/fewer GPUs to the training process. We'll initialize the weights layers in the network using Xavier initialization (**Line 87**) and allow the network to train for a maximum of 90 epochs. As our experiments will demonstrate, we'll apply early stopping at epoch 80 to reduce overfitting.

In the following code block we define our callbacks and evaluation metrics:

```
94  # initialize the callbacks and evaluation metrics
95  batchEndCBs = [mx.callback.Speedometer(batchSize, 250)]
96  epochEndCBs = [mx.callback.do_checkpoint(checkpointsPath)]
97  metrics = [mx.metric.Accuracy(), mx.metric.TopKAccuracy(top_k=5),
98      mx.metric.CrossEntropy()]
```

And finally we can train our network:

```
100 # train the network
101 print("[INFO] training network...")
102 model.fit(
103     X=trainIter,
104     eval_data=valIter,
105     eval_metric=metrics,
106     batch_end_callback=batchEndCBs,
107     epoch_end_callback=epochEndCBs)
```

Again, just as in all previous ImageNet chapters in this book, our `train_squeezenet.py` script is nearly identical to all other `train_*.py` scripts – the only real difference is in:

1. The configuration import file.
2. The model import file.
3. The model instantiation.
4. Updates to the SGD optimizer.
5. Any weight initializations.

Using this template, you can easily create your own training script in a manner of minutes, provided that you have already coded your network architecture.

10.3 Evaluating SqueezeNet

To evaluate SqueezeNet, we'll be using the `test_squeezenet.py` script mentioned in our project structure above. Again, just as in all other ImageNet experiments in the book, this script is *identical* to `test_alexnet.py` and all other `test_*.py` scripts used to evaluate a given network architecture. Since these scripts are identical, I will not be reviewing `test_squeezenet.py` here. Please consult Chapter 6 on `test_alexnet.py` for a thorough review of the code.

Additionally, you can use the downloads portion of this book to inspect the project and see the contents of `test_squeezenet.py`. Again, the contents of these files are *identical* as they are part of our framework for training and evaluating CNNs trained on ImageNet. I simply provided a separate `test_squeezenet.py` script here just in case you wished to perform your own modifications.

10.4 SqueezeNet Experiments

In 2016, Iandola et al. made the claim that SqueezeNet could obtain AlexNet-level accuracy with $50x$ fewer parameters. I would define AlexNet accuracy in the range of 55-58% rank-1 accuracy. Thus, if our version of SqueezeNet can fall inside this range of rank-1 accuracy, I will call our reproduction of the experiment a success.

Just like in previous chapters, I'll explain my initial baseline experiments, discuss what worked (and what didn't), and how I modified any parameters before the next experiment. In total, I required four experiments to replicate the results of Iandola et al. You can find the results of my experiments below – use these results to help you when performing experiments of your own.

It's not just the *end result* that matters, it's the *scientific method* of devising an experiment, running it, gathering the results, and tuning the parameters that is critical. To become a true deep learning expert, you need to study the *path* that leads to an optimal CNN, not just the end result. Reading case studies such as these will help you develop this skill.

10.4.1 SqueezeNet: Experiment #1

In my first SqueezeNet experiment I attempted to replicate the work of Iandola et al. by using their *exact* architecture and optimizer parameters. Instead of using ELUs as reported in the implementation section above, I used ReLUs. I also used an initial learning rate of $4e-2$, as recommended by the original paper. To start the training process, I executed the following command:

```
$ python train_squeezenet.py --checkpoints checkpoints --prefix squeezenet
```

I closely monitored my training process, but was noticing *significant volatility* in the training process, especially during epoch 15 and epochs in the early 20's (Figure 10.3, *top-left*). In the latter case, accuracy plummeted all the way down to 0.9%, but was able to recover again. After epoch 25 I decided to lower the learning rate from $4e-2$ to $4e-3$ to see if the lower learning rate helped stabilize the network:

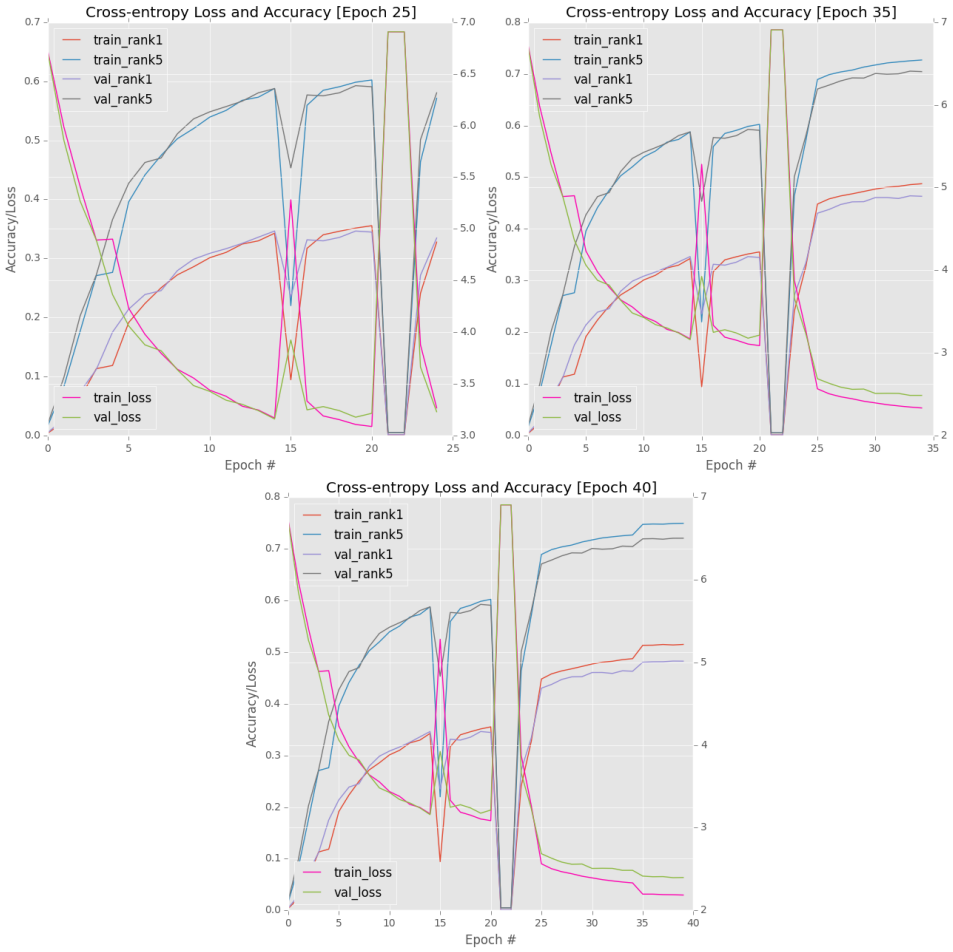


Figure 10.3: **Top-left:** Training SqueezeNet on ImageNet with an initial learning rate of $4e-2$ (suggested by Iandola et al. [30]) demonstrates dramatic volatility. **Top-right:** Reducing the learning rate from $4e-2$ to $4e-3$ helps reduce the volatility; however, learning is starting to stagnate. **Bottom:** Adjusting α to $4e-4$ leads to total stagnation.

```
$ python train_squeezenet.py --checkpoints checkpoints --prefix squeezenet \
    --start-epoch 25
```

Lowering the learning rate helped stabilize the learning curve; however, validation loss started declining at a much slower rate (Figure 10.3, *top-right*). At epoch 35 I stopped training and adjusted the learning rate from $4e-3$ to $4e-4$, mainly just to validate my intuition that this change in α would totally stagnate learning:

```
$ python train_squeezenet.py --checkpoints checkpoints --prefix squeezenet \
    --start-epoch 35
```

In this case, my intuition was correct – lowering the learning rate stagnated results completely (Figure 10.3, *bottom*). At epoch 40 I killed off the experiment. Given both the volatility and stagnation, it was clear that the initial learning rate needed to be adjusted (and likely a few other hyperparameters).

However, this experiment was not for nothing. Most importantly, I was able to obtain a baseline accuracy. After epoch 40, SqueezeNet was obtaining 48.93% rank-1 and 72.07% rank-5 accuracy on the validation set. This is a far cry from AlexNet-level accuracy, so we clearly have some work to do.

10.4.2 SqueezeNet: Experiment #2

Given that training SqueezeNet with a $4e-2$ learning rate was so volatile, I decided to reduce the learning rate to a more standard $1e-2$. I also kept ReLU activations and did not swap them out for ELUs (that will come later). I started training SqueezeNet using the following command:

```
$ python train_squeezenet.py --checkpoints checkpoints --prefix squeezenet
```

I then allowed SqueezeNet to train for 50 epochs at the $1e-2$ learning rate (Figure 10.4, *top-left*).

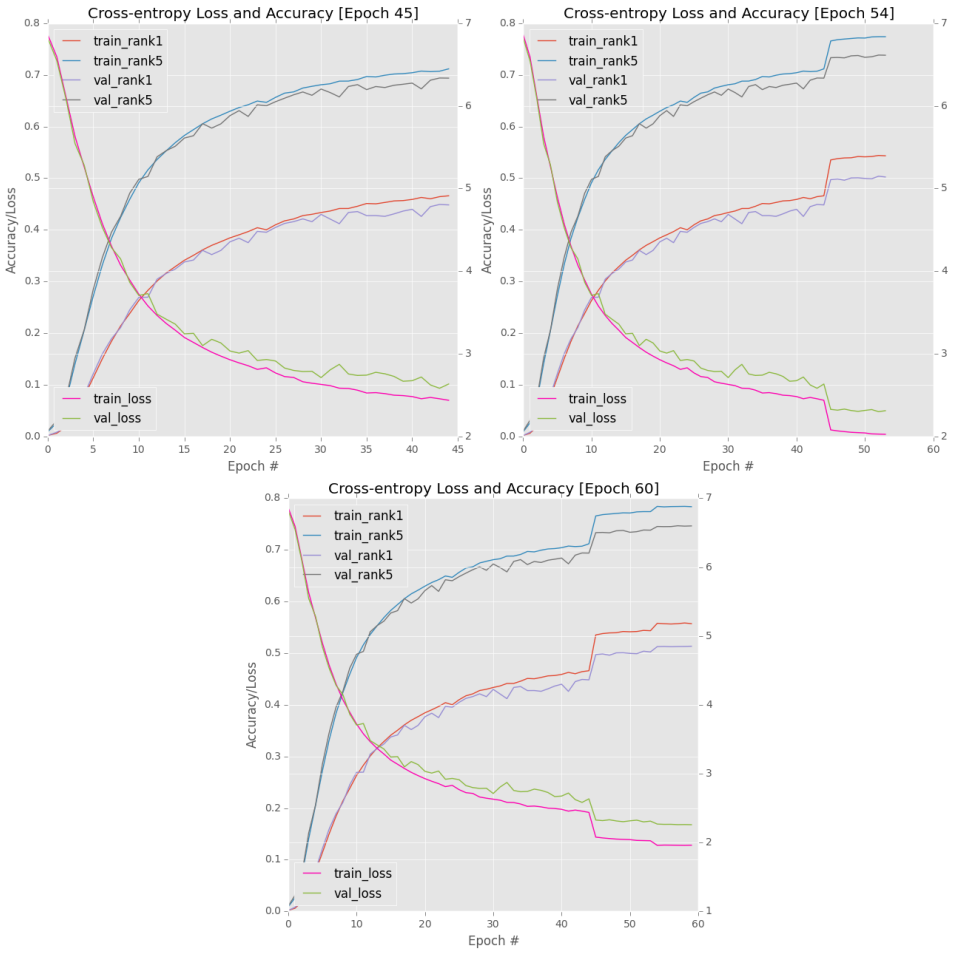


Figure 10.4: **Top-left:** Training SqueezeNet with an initial of $\alpha = 1e-2$ is stable but slow going. **Top-right:** Lowering the learning rate to $1e-3$ leads to an increase in accuracy/drop in loss; however, learning quickly stagnates. **Bottom:** Learning has totally stagnated with $\alpha = 1e-4$.

There are two key takeaways from this plot:

1. Learning is *much* more smooth and less volatile. SqueezeNet is not randomly dropping down to 0% classification accuracy.
2. Similarly, the training and validation curves mimic each other near exactly until epoch 30 where we see divergence.

However, past epoch 50, the divergence continues and there is stagnation in the validation loss/accuracy. Because of this stagnation, I decided to `ctrl + c` out of the experiment and restart my training from epoch 45 with a lower learning rate of $1e-3$:

```
$ python train_squeezenet.py --checkpoints checkpoints --prefix squeezenet \
    --start-epoch 45
```

As you can see, loss immediately drops and accuracy increases, as the optimizer lands in lower areas of loss in the loss landscape (Figure 10.4, *top-right*). However, this increase in accuracy stagnates quickly so I once again `ctrl + c` out of the experiment and restart training at epoch 55 with a $1e-4$ learning rate:

```
$ python train_squeezenet.py --checkpoints checkpoints --prefix squeezenet \
    --start-epoch 55
```

Training continues for another five epochs before I officially stop the experiment as there are no further gains to be had (Figure 10.4, *bottom*). Looking at my validation data, I found that SqueezeNet was now obtaining 51.30% rank-1 and 74.60% rank-5 accuracy on the validation set, a marked improvement from the previous experiment; however, we are *still* not yet at AlexNet-level accuracy.

10.4.3 SqueezeNet: Experiment #3

My third experiment is an excellent example of a *failed* experiment – one that sounds good in your head, but when you actually apply it, fails miserably. Given that micro-architectures such as GoogLeNet and ResNet benefit from having batch normalization layers, I decided to update the SqueezeNet architecture to include batch normalizations after every ReLU, an example of which can be seen below:

```
46          # Block #1: CONV => RELU => POOL
47          conv_1 = mx.sym.Convolution(data=data, kernel=(7, 7),
48               stride=(2, 2), num_filter=96)
49          relu_1 = mx.sym.Activation(data=conv_1, act_type="relu")
50          bn_1 = mx.sym.BatchNorm(data=relu_1)
51          pool_1 = mx.sym.Pooling(data=bn_1, kernel=(3, 3),
52               stride=(2, 2), pool_type="max")
```

I then started training once again:

```
$ python train_squeezenet.py --checkpoints checkpoints --prefix squeezenet
```

It took less than 20 epochs to realize that this experiment was not going to turn out well (Figure 10.5). While SqueezeNet started off with a higher initial accuracy during the earlier epochs (close to 10% rank-1 and 20% rank-5), these benefits were quickly lost. The network was slow to train

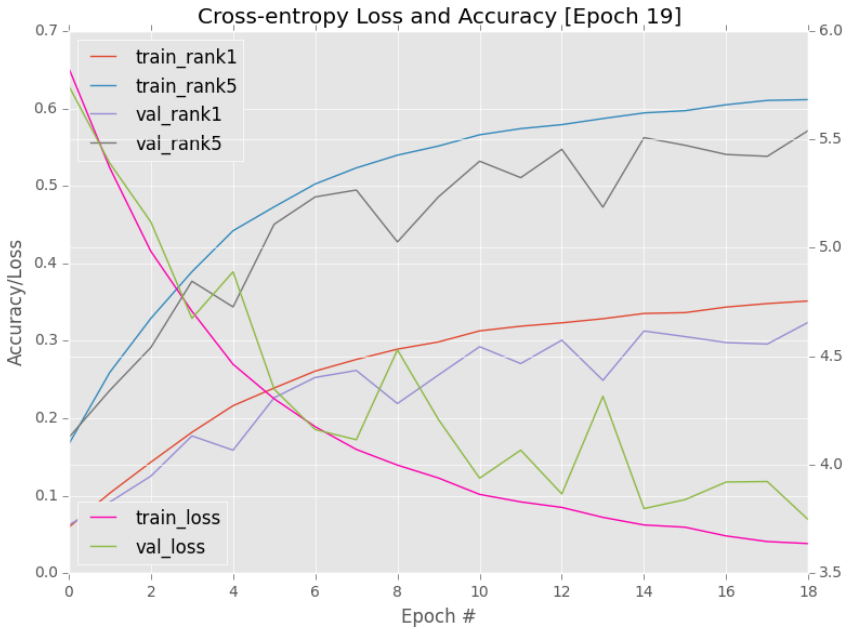


Figure 10.5: Adding batch normalization layers to SqueezeNet was a total failure and I quickly killed the experiment.

and furthermore the validation data reported volatility. Based on Experiment #2, I was expecting to have $> 40\%$ rank-1 accuracy by the time I reached epoch 20 – as Figure 10.5 above demonstrates, this result was not going to happen.

I killed off this experiment and noted in my lab journal that batch normalization layers *do not* help the SqueezeNet architecture, something I found very surprising. I then removed all batch normalization layers from SqueezeNet and moved to my next experiment.

10.4.4 SqueezeNet: Experiment #4

At this point, my best result had come from Experiment #2 (51.30% rank-1 and 74.60% rank-5), but I couldn't break my way into the AlexNet-level accuracy of 55-58% accuracy. In an attempt to reach this point, I decided to perform my old trick of swapping out ReLUs for ELUs:

```
46          # Block #1: CONV => RELU => POOL
47          conv_1 = mx.sym.Convolution(data=data, kernel=(7, 7),
48               stride=(2, 2), num_filter=96)
49          relu_1 = mx.sym.LeakyReLU(data=conv_1, act_type="elu")
50          pool_1 = mx.sym.Pooling(data=relu_1, kernel=(3, 3),
51               stride=(2, 2), pool_type="max")
```

Training was then started using the following command:

```
$ python train_squeezenet.py --checkpoints checkpoints --prefix squeezenet
```

At first, I was a bit worried regarding the learning curve – swapping out ReLUs for ELUs I was expecting slightly higher than 40% rank-1 accuracy by epoch 20; however, I let training continue as

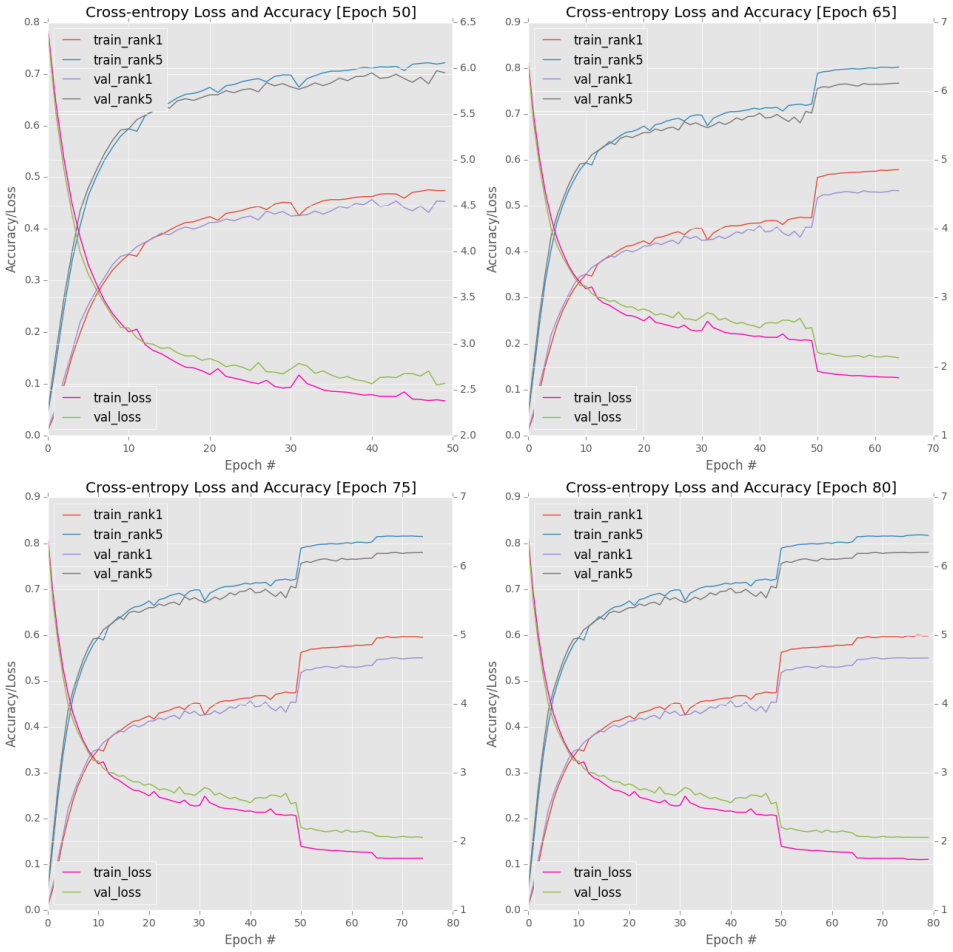


Figure 10.6: **Top-left:** Training SqueezeNet on ImageNet with ELUs instead of ReLUs at $\alpha = 1e-2$. **Top-right:** Lowering α to $1e-3$ gives us a nice boost in accuracy. **Bottom-left:** Another small boost is obtained with $\alpha = 1e-4$. **Bottom-right:** Learning has essentially stagnated once we reach $\alpha = 1e-5$.

learning was quite stable. This turned out to be a good decision as it allowed SqueezeNet to *train for longer* without having to update the learning rate (Figure 10.6, *top-left*).

By the time I reached epoch 50, I started to notice stagnation in the validation accuracy/loss (and even a few accompanying dips), so I stopped training, lowered the learning rate from $1e-2$ to $1e-3$, and then continued training:

```
$ python train_squeezenet.py --checkpoints checkpoints --prefix squeezenet \
    --start-epoch 50
```

As we can see, the jump in validation accuracy (and drop in validation loss) is quite dramatic, more pronounced than the second experiment – this jump alone put me over the 51% rank-1 mark (Figure 10.6, *top-right*). At this point, I was confident that using ELUs was a good choice. I allowed training to continue for 15 epochs where I once again stopped training, reduced the learning rate from $1e-3$ to $1e-4$, and restarted training:

```
$ python train_squeezenet.py --checkpoints checkpoints --prefix squeezenet \
    --start-epoch 65
```

Again, we can see another jump in validation accuracy and drop in validation loss, although less pronounced than the previous time (Figure 10.6, *bottom-left*). SqueezNet was reaching $\approx 54\%$ classification accuracy after epoch 75, but there was definite stagnation. I decided to try a $1e-5$ learning rate for five epochs just to see if it made a difference, but at this point the learning rate was too low for the optimizer to find any lower areas of loss in the loss landscape:

```
$ python train_squeezenet.py --checkpoints checkpoints --prefix squeezenet \
    --start-epoch 75
```

At the end of epoch 80, I stopped training entirely and evaluated SqueezeNet (Figure 10.6, *bottom-right*). On the validation set, this iteration of SqueezeNet reached 54.49% rank-1 and 77.98% rank-5 accuracy, which is *very* close to AlexNet-levels. Given this encouraging result, I switched over to the test set and evaluated:

```
$ python test_squeezenet.py --checkpoints checkpoints --prefix squeezenet \
    --epoch 80
[INFO] loading model...
[INFO] predicting on test data...
[INFO] rank-1: 57.01%
[INFO] rank-5: 79.45%
```

As the results demonstrate, SqueezeNet was able to obtain $\mathbf{57.01}\%$ rank-1 and $\mathbf{79.45}\%$ rank-5 accuracy, *well within* AlexNet-level results.

Through this experiment, I was able to successfully replicate the results of Iandola et al., although I must admit that much of this success hinged on replacing their original reporting learning rate of $4e-2$ with $1e-2$. After doing a bit of research online regarding the experience other deep learning practitioners had with training SqueezeNet from scratch, most others report that $4e-2$ was *far* too large, sometimes requiring them to lower the learning rate to $1e-3$ to obtain any traction.

Experiments such as these go to show you how challenging it can be to replicate the results of a published work. Depending on the deep learning library you use and the versions of CUDA/cuDNN running on your system, you may get different results even if your implementation is identical to the authors'. Furthermore, some deep learning libraries implement layers differently, so there may be underlying parameters that you are unaware of this.

A great example of this is the ELU activation layer:

- In Keras, the ELU activation uses a default α value of 1.0.
- But in mxnet, the ELU α value defaults to 0.25.

It's subtle, but small variations like these can really add up. Instead of becoming disgruntled and frustrated that your implementation of a given network doesn't achieve the *exact* reported results of the author, do a little more research first. See if you can determine what *libraries* and what *assumptions* they made when training their network. From there, start experimenting. Note what works and what doesn't – *and be sure to log these results in a journal!* After an experiment completes, revisit the results and examine what you can tweak to boost accuracy further. In most cases, this will be your learning rate, regularization, weight initializations, and activation functions.

10.5 Summary

In this chapter, we learned how to train the SqueezeNet architecture from scratch on the ImageNet dataset. Furthermore, we were able to successfully reproduce the results of Iandola et al., successfully obtaining AlexNet-level accuracy – $\mathbf{57.01}\%$ rank-1 and $\mathbf{79.45}\%$ rank-5 on the testing set, respectively. To achieve this result, we needed to reduce the $4e-2$ learning rate originally suggested by Iandola et al. and use a smaller learning rate of $1e-2$. This smaller learning rate helped reduce volatility in the model and ultimately obtain higher classification.

Secondly, swapping out ReLUs for ELUs allowed us to train the network for longer with less stagnation. Ultimately, it was the activation function swap that allowed us to obtain the AlexNet-level results.

As I've mentioned in previous chapters, always start with standard ReLU activation function to (1) ensure your model can be trained properly and (2) obtain a baseline. Then, go back and tweak other hyperparameters to the network, including learning rate scheduling, regularization/weight decay, weight initialization, and even batch normalization. Once you have optimized your network as far as you think you can, swap out ReLUs for ELUs to (normally) obtain a small boost in classification accuracy. The only exception to this rule is if you are training a very deep CNN and require the usage of MSRA/He et al. initialization – in this case, as soon as you make the swap to MSRA initialization, you should also consider swapping out standard ReLUs for PReLUs.

This chapter concludes our discussion on training state-of-the-art Convolutional Neural Networks on the challenging ImageNet dataset. Be sure to practice and experiment with these networks as much as you can. Devise experiments of your own and run experiments *not* included in this book. Explore various hyperparameters and the effects they have on training a given network architecture.

Throughout this book, I have consolidated my ImageNet experiments so they can easily be read and digested. In some cases, I needed to run 20+ experiments (AlexNet being a prime example) before obtaining my best results. I then selected the most important experiments to include in this book to demonstrate the right (and wrong) turns I made.

The same will be true for you as well. Don't be discouraged if many of your experiments aren't immediately obtaining excellent results, even if you feel you are applying *identical* architectures and parameters as the original authors – ***this trial and error is part of the process***.

Deep learning experimentation is *iterative*. Develop a hypothesis. Run the experiment. Evaluate the results. Tweak the parameters. And run another experiment.

Keep in mind that the world's top deep learning experts perform *tens* to *hundreds* of experiments *per project* – the same will be true for you as you develop your ability to train deep neural networks on challenging datasets. This skill hinges right on the cusp of "science" and "art" – and the only way to obtain this much-needed skill is *practice*.

In the remaining chapters in the *ImageNet Bundle*, we'll look at a variety of case studies, including emotion/facial expression recognition, predicting and correcting image orientation, classifying the make and model of a vehicle, and predicting the age and gender of a person in a video. These case studies will help you further improve your skills as a deep learning practitioner.

11. Case Study: Emotion Recognition

In this chapter, we are going to tackle Kaggle's Facial Expression Recognition challenge [31] and claim a top-5 spot on the leaderboard. To accomplish this task, we are going to train a VGG-like network from scratch on the training data, while taking into account that our network needs to be *small enough* and *fast enough* to run in real-time on our CPU. Once our network is trained (and we have claimed our top-5 spot on the leaderboard), we'll write some code to load the serialized model from disk and apply it to a video stream to detect emotions in real-time.

However, before we get started, it's important to understand that as humans, our emotions are in a constant fluid state. We are never 100% happy or 100% sad. Instead, our emotions mix together. When experiencing "surprise" we might also be feeling "happiness" (such as a surprise birthday party) or "fright" (if the surprise is not a welcome one). And even during the "scared" emotion, we might feel hints of "anger" as well.

When studying emotion recognition, it's important to not focus on a *single class label* (as we sometimes do in other classification problems). Instead, it's much more advantageous for us to look at the *probability* of each emotion and characterize the *distribution*. As we'll see later in this chapter, examining the distribution of emotion probabilities provides us with a more accurate gauge of emotion prediction than simply picking a *single* emotion with the highest probability.

11.1 The Kaggle Facial Expression Recognition Challenge

The Kaggle Emotion and Facial Expression Recognition challenge training dataset consists of 28,709 images, each of which are 48×48 grayscale images (Figure 11.1). The faces have been automatically aligned such that they are approximately the same size in each image. Given these images, our goal is to categorize the emotion expressed on each face into seven distinct classes: *angry*, *disgust*, *fear*, *happy*, *sad*, *surprise*, and *neutral*.

11.1.1 The FER13 Dataset

The dataset used in the Kaggle competition was aggregated by Goodfellow et al. in their 2013 paper, *Challenges in Representation Learning: A report on three machine learning contests* [32].

Figure 11.1: A sample of the facial expressions inside the Kaggle: Facial Expression Recognition Challenge. We will train a CNN to recognize and identify each of these emotions. This CNN will also be able to run in *real-time* on your CPU, enabling you to recognize emotions in video streams.

This facial expression dataset is called the *FER13 dataset* and can be found at the official Kaggle competition page and downloading the `fer2013.tar.gz` file:

http://pyimg.co/a2soy

The `.tar.gz` archive of the dataset is $\approx 92MB$, so make sure you have a decent internet connection before downloading it. After downloading the dataset, you'll find a file named `fer2013.csv` with with three columns:

- `emotion`: The class label.
- `pixels`: A flattened list of $48 \times 48 = 2,304$ grayscale pixels representing the face itself.
- `usage`: Whether the image is for `Training`, `PrivateTest` (validation), or `PublicTest` (testing).

Our goal is to now take this `.csv` file and convert it to HDF5 format so we can more easily train a Convolutional Neural Network on top of it.

After I unpacked the `fer2013.tar.gz` file I set up the following directory structure for the project:

```
--- fer2013
|     |--- fer2013
|     |--- hdf5
|     |--- output
```

Inside the `fer2013` directory, I have included the contents of the unarchived `fer2013.tar.gz` file (i.e., the dataset itself). The `hdf5` directory will contain the training, testing, and validation splits. Finally, we'll save any training logs/plots to the `output` directory.

11.1.2 Building the FER13 Dataset

Let's go ahead and setup our directory structure for this emotion recognition project:

```
--- emotion_recognition
|    |--- config
|    |    |--- __init__.py
|    |    |--- emotion_config.py
|    |--- build_dataset.py
|    |--- emotion_detector.py
|    |--- haarcascade_frontface_default.xml
|    |--- test_recognizer.py
|    |--- train_recognizer.py
```

As in previous experiments, we'll create a directory named `config` and turn it into a Python sub-module by placing a `__init__.py` field inside of it. Inside `config`, we'll then create a file named `emotion_config.py` – this file is where we'll store any configuration variables, including paths to the input dataset, output HDF5 files, and batch sizes.

The `build_dataset.py` will be responsible for ingesting the `fer2013.csv` dataset file and outputting set a set of HDF5 files; one for each of the training, validation, and testing splits, respectively. We'll train our CNN to recognize various emotions using `train_recognizer.py`. Similarly, the `test_recognizer.py` script will be used to evaluate the performance of CNN.

Once we are happy with the accuracy of our model, we'll then move on to implementing the `emotion_detector.py` script to:

1. Detect faces in real-time (as in our smile detector chapter in the *Starter Bundle*).
2. Apply our CNN to recognize the most dominant emotion and display the probability distribution for each emotion.

Most importantly, this CNN will be able to run and detect facial expressions in *real-time* on our CPU. Let's go ahead and review the `emotion_config.py` file now:

```
1  # import the necessary packages
2  from os import path
3
4  # define the base path to the emotion dataset
5  BASE_PATH = "/raid/datasets/fer2013"
6
7  # use the base path to define the path to the input emotions file
8  INPUT_PATH = path.sep.join([BASE_PATH, "fer2013/fer2013.csv"])
```

On **Line 5** we define the `BASE_PATH` to our root project directory. This location is where the input FER13 dataset will live, along with the output HDF5 dataset files, logs, and plots. **Line 8** then defines the `INPUT_PATH` to the actual `fer2013.csv` file itself.

Let's also define the number of classes in the FER13 dataset:

```
10 # define the number of classes (set to 6 if you are ignoring the
11 # "disgust" class)
12 # NUM_CLASSES = 7
13 NUM_CLASSES = 6
```

In total, there are seven classes in FER13: *angry*, *disgust*, *fear*, *happy*, *sad*, *surprise*, and *neutral*. However, there is heavy class imbalance with the "disgust" class, as it has only 113 image samples (the rest have over 1,000 images per class). After doing some research, I came across the Mememoji project [33] which suggests merging both "disgust" and "anger" into a single class

(as the emotions are visually similar), thereby turning FER13 into a 6 class problem. In this chapter we will investigate emotion recognition using both six and seven classes; however, when it's time to actually *deploy* and use our CNN to classify emotions, we'll reduce the number of emotions to six to better improve the results.

Since we'll be converting the `fer2013.csv` file into a series of HDF5 datasets for training, validation, and testing, we need to define the paths to these output HDF5 files:

```
15 # define the path to the output training, validation, and testing
16 # HDF5 files
17 TRAIN_HDF5 = path.sep.join([BASE_PATH, "hdf5/train.hdf5"])
18 VAL_HDF5 = path.sep.join([BASE_PATH, "hdf5/val.hdf5"])
19 TEST_HDF5 = path.sep.join([BASE_PATH, "hdf5/test.hdf5"])
```

Finally, we'll initialize the batch size when training our CNN along with the output directory where any logs or plots will be stored:

```
21 # define the batch size
22 BATCH_SIZE = 128
23
24 # define the path to where output logs will be stored
25 OUTPUT_PATH = path.sep.join([BASE_PATH, "output"])
```

Now that our configuration file is created, we can actually build the dataset:

```
1  # import the necessary packages
2  from config import emotion_config as config
3  from pyimagesearch.io import HDF5DatasetWriter
4  import numpy as np
5
6  # open the input file for reading (skipping the header), then
7  # initialize the list of data and labels for the training,
8  # validation, and testing sets
9  print("[INFO] loading input data...")
10 f = open(config.INPUT_PATH)
11 f.__next__() # f.next() for Python 2.7
12 (trainImages, trainLabels) = ([], [])
13 (valImages, valLabels) = ([], [])
14 (testImages, testLabels) = ([], [])
```

On **Line 2** we import our `emotion_config` file that we just created. **Line 3** then imports our `HDF5DatasetWriter` which we have used many times in this book to convert an input set of images to an HDF5 dataset.

Line 10 opens a pointer to the input `fer2013.csv` file. By calling the `.next` method of the file pointer, we can skip to the next line, allowing us to skip the header of the CSV file. **Lines 12-14** then initialize lists of images and labels for the training, validation, and testing sets, respectively.

We are now ready to start building our data splits:

```
16 # loop over the rows in the input file
17 for row in f:
18     # extract the label, image, and usage from the row
```

```
19      (label, image, usage) = row.strip().split(",")
20      label = int(label)
21
22      # if we are ignoring the "disgust" class there will be 6 total
23      # class labels instead of 7
24      if config.NUM_CLASSES == 6:
25          # merge together the "anger" and "disgust classes
26          if label == 1:
27              label = 0
28
29          # if label has a value greater than zero, subtract one from
30          # it to make all labels sequential (not required, but helps
31          # when interpreting results)
32          if label > 0:
33              label -= 1
```

On **Line 17** we start looping over each of the rows in the input file. **Line 19** takes the `row` and splits it into a 3-tuple of the image `label`, the raw pixel intensities of the `image`, along with the `usage` (i.e., training, testing, or validation). By default, we'll assume that we are treating FER13 as a 7-class classification problem; however, in the case that we wish to merge the anger and disgust class together (**Line 24**), we need to change disgust label from a `1` to a `0` (**Lines 26 and 27**). We'll also subtract a value of `1` from every label on **Lines 32 and 33** to ensure that each class label is *sequential* – this subtraction isn't required but it helps when interpreting our results.

At this point our image is just a string of integers. We need to take this string, split it into a list, convert it to an unsigned 8-bit integer data type, and reshape it to a 48×48 grayscale image:

```
35      # reshape the flattened pixel list into a 48x48 (grayscale)
36      # image
37      image = np.array(image.split(" "), dtype="uint8")
38      image = image.reshape((48, 48))
```

Keep in mind that each image column is a list of 2,304 integers. These 2,304 integers represent the square 48×48 image. We can perform this reshaping on **Line 38**.

The last step in parsing the `fer2013.csv` file is to simply check the `usage` and assign the `image` and `label` to the respective training, validation, or testing lists:

```
40      # check if we are examining a training image
41      if usage == "Training":
42          trainImages.append(image)
43          trainLabels.append(label)
44
45      # check if this is a validation image
46      elif usage == "PrivateTest":
47          valImages.append(image)
48          valLabels.append(label)
49
50      # otherwise, this must be a testing image
51      else:
52          testImages.append(image)
53          testLabels.append(label)
```

From here, our dataset creation code starts to become more familiar:

```
55 # construct a list pairing the training, validation, and testing
56 # images along with their corresponding labels and output HDF5
57 # files
58 datasets = [
59      (trainImages, trainLabels, config.TRAIN_HDF5),
60      (valImages, valLabels, config.VAL_HDF5),
61      (testImages, testLabels, config.TEST_HDF5)]
```

Here we are simply initializing the `datasets` list. Each entry in the list is a 3-tuple of the raw images, labels, and output HDF5 path.

The last step is to loop over each of the training, validation, and testing sets:

```
63 # loop over the dataset tuples
64 for (images, labels, outputPath) in datasets:
65      # create HDF5 writer
66      print("[INFO] building {}...".format(outputPath))
67      writer = HDF5DatasetWriter((len(images), 48, 48), outputPath)
68
69      # loop over the image and add them to the dataset
70      for (image, label) in zip(images, labels):
71           writer.add([image], [label])
72
73      # close the HDF5 writer
74      writer.close()
75
76 # close the input file
77 f.close()
```

Line 67 instantiates a `HDF5DatasetWriter` for the dataset. We then loop over the pairs of `images` and `labels`, writing them to disk in HDF5 format (**Lines 70 and 71**).

To build the FER2013 dataset for emotion recognition, simply execute the following command:

```
$ python build_dataset.py
[INFO] loading input data...
[INFO] building /raid/datasets/fer2013/hdf5/train.hdf5...
[INFO] building /raid/datasets/fer2013/hdf5/val.hdf5...
[INFO] building /raid/datasets/fer2013/hdf5/test.hdf5...
```

After the command finishes executing, you can validate that the HDF5 files have been generated by examining the contents of the directory where you instructed your HDF5 files to be stored inside `emotion_config.py`:

```
$ ls -l fer2013/hdf5
total 646268
-rw-rw-r-- 1 adrian adrian  66183304 Aug 29 08:25 test.hdf5
-rw-rw-r-- 1 adrian adrian 529396104 Aug 29 08:25 train.hdf5
-rw-rw-r-- 1 adrian adrian  66183304 Aug 29 08:25 val.hdf5
```

Notice how I have three files: `train.hdf5`, `val.hdf5`, and `test.hdf5` – these are the files I will use when training and evaluating my network on the FER2013 dataset.

Layer Type	Output Size	Filter Size / Stride
INPUT IMAGE	$48 \times 48 \times 1$	
CONV	$48 \times 48 \times 32$	$3 \times 3, K = 32$
CONV	$48 \times 48 \times 32$	$3 \times 3, K = 32$
POOL	$24 \times 24 \times 32$	2×2
CONV	$24 \times 24 \times 64$	$3 \times 3, K = 64$
CONV	$24 \times 24 \times 64$	$3 \times 3, K = 64$
POOL	$12 \times 12 \times 64$	2×2
CONV	$12 \times 12 \times 128$	$3 \times 3, K = 128$
CONV	$12 \times 12 \times 128$	$3 \times 3, K = 128$
POOL	$6 \times 6 \times 128$	2×2
FC	64	
FC	64	
FC	6	
SOFTMAX	6	

Table 11.1: A table summary of the EmotionVGGnet architecture. Output volume sizes are included for each layer, along with convolutional filter size/pool size when relevant.

11.2 Implementing a VGG-like Network

The network we are going to implement to recognize various emotions and facial expressions is inspired by the family of VGG networks:

1. The `CONV` layers in the network will *only* be 3×3.
2. We'll double the number of filters learned by each `CONV` layer the deeper we go in the network.

To aid in training of the network, we'll apply some *a priori* knowledge gained from experimenting with VGG and ImageNet in Chapter 9:

1. We should initialize our `CONV` and `FC` layers using the MSRA/He et al. method – doing so will enable our network to learn faster.
2. Since ELUs and PReLUs have been shown to boost classification accuracy throughout all of our experiments, let's simply start with an ELU instead of a ReLU.

I have included a summary of the network, named EmotionVGGNet, in Table 11.1. After every `CONV` layer, we will apply an activation followed by a batch normalization – these layers were purposely left out of the table to save space. Let's go ahead and implement the `EmotionVGGNet` class now. Add a new file named `emotionvggnet.py` to the `nn.conv` sub-module of `pyimagesearch`:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- io
|    |--- nn
|    |    |--- __init__.py
|    |    |--- conv
|    |    |    |--- __init__.py
...
|    |    |    |--- emotionvggnet.py
...
|    |--- preprocessing
|    |--- utils
```

From there, open `emotionvggnet.py` and start by importing our required Python packages:

```
1  # import the necessary packages
2  from keras.models import Sequential
3  from keras.layers.normalization import BatchNormalization
4  from keras.layers.convolutional import Conv2D
5  from keras.layers.convolutional import MaxPooling2D
6  from keras.layers.advanced_activations import ELU
7  from keras.layers.core import Activation
8  from keras.layers.core import Flatten
9  from keras.layers.core import Dropout
10 from keras.layers.core import Dense
11 from keras import backend as K
```

All of these imports are standard when building Convolutional Neural Networks with Keras, so we can skip an in-depth discussion of them (not to mention, you've seen these imports many times before in this book).

From there, we can define the `build` method of `EmotionVGGNet`:

```
13 class EmotionVGGNet:
14     @staticmethod
15     def build(width, height, depth, classes):
16         # initialize the model along with the input shape to be
17         # "channels last" and the channels dimension itself
18         model = Sequential()
19         inputShape = (height, width, depth)
20         chanDim = -1
21
22         # if we are using "channels first", update the input shape
23         # and channels dimension
24         if K.image_data_format() == "channels_first":
25             inputShape = (depth, height, width)
26             chanDim = 1
```

Line 18 initializes the `model` object that we'll be adding layers to. Since the VGG family of networks sequentially applies layers one right after the other, we can use the `Sequential` class here. We'll also assume "channels last" ordering (**Lines 19 and 20**), but if we are instead using "channels first" ordering, we can update the input shape and channel dimension axis as well (**Lines 24-26**).

Each block in convolution block in EmotionVGGNet will consist of `(CONV => RELU => BN) * 2 => POOL` layer sets. Let's define the first block now:

```
28         # Block #1: first CONV => RELU => CONV => RELU => POOL
29         # layer set
30         model.add(Conv2D(32, (3, 3), padding="same",
31             kernel_initializer="he_normal", input_shape=inputShape))
32         model.add(ELU())
33         model.add(BatchNormalization(axis=chanDim))
34         model.add(Conv2D(32, (3, 3), kernel_initializer="he_normal",
35             padding="same"))
36         model.add(ELU())
37         model.add(BatchNormalization(axis=chanDim))
```

```
38            model.add(MaxPooling2D(pool_size=(2, 2)))
39            model.add(Dropout(0.25))
```

The first CONV layer will learn 32 3×3 filters. We'll then apply an ELU activation followed by batch normalization. Similarly, the second CONV layer applies the same pattern, learning 32 3×3 filters, followed by an ELU and batch normalization. We then apply max pooling and then a dropout layer with probability 25%.

The second block in EmotionVGGNet is identical to the first, only now we are doubling the number of filters in the CONV layers to 64 rather than 32:

```
41            # Block #2: second CONV => RELU => CONV => RELU => POOL
42            # layer set
43            model.add(Conv2D(64, (3, 3), kernel_initializer="he_normal",
44                 padding="same"))
45            model.add(ELU())
46            model.add(BatchNormalization(axis=chanDim))
47            model.add(Conv2D(64, (3, 3), kernel_initializer="he_normal",
48                 padding="same"))
49            model.add(ELU())
50            model.add(BatchNormalization(axis=chanDim))
51            model.add(MaxPooling2D(pool_size=(2, 2)))
52            model.add(Dropout(0.25))
```

Moving on to the third block, we again apply the same pattern, now increasing the number of filters from 64 to 128 – as we get deeper in the CNN, the more filters we learn:

```
54            # Block #3: third CONV => RELU => CONV => RELU => POOL
55            # layer set
56            model.add(Conv2D(128, (3, 3), kernel_initializer="he_normal",
57                 padding="same"))
58            model.add(ELU())
59            model.add(BatchNormalization(axis=chanDim))
60            model.add(Conv2D(128, (3, 3), kernel_initializer="he_normal",
61                 padding="same"))
62            model.add(ELU())
63            model.add(BatchNormalization(axis=chanDim))
64            model.add(MaxPooling2D(pool_size=(2, 2)))
65            model.add(Dropout(0.25))
```

Next, we need to construct our first fully-connected layer set:

```
67            # Block #4: first set of FC => RELU layers
68            model.add(Flatten())
69            model.add(Dense(64, kernel_initializer="he_normal"))
70            model.add(ELU())
71            model.add(BatchNormalization())
72            model.add(Dropout(0.5))
```

Here we learn 64 hidden nodes, followed by applying an ELU activation function and batch normalization. We'll apply a second FC layer in the same manner:

```
74            # Block #6: second set of FC => RELU layers
75            model.add(Dense(64, kernel_initializer="he_normal"))
76            model.add(ELU())
77            model.add(BatchNormalization())
78            model.add(Dropout(0.5))
```

Finally, we'll apply a `FC` layer with the supplied number of `classes` along with a softmax classifier to obtain our output class label probabilities:

```
80            # Block #7: softmax classifier
81            model.add(Dense(classes, kernel_initializer="he_normal"))
82            model.add(Activation("softmax"))
83
84            # return the constructed network architecture
85            return model
```

Now that we have implemented our `EmotionVGGNet` class, let's go ahead and train it.

11.3 Training Our Facial Expression Recognizer

In this section, we will review how to train EmotionVGGNet from scratch on the FER2013 dataset. We'll start with a review of the `train_recognizer.py` script used to train FER2013, followed by examining a set of experiments I performed to maximize classification accuracy. To get started, create a new file named `train_recognizer.py` and insert the following code:

```
1  # set the matplotlib backend so figures can be saved in the background
2  import matplotlib
3  matplotlib.use("Agg")
4
5  # import the necessary packages
6  from config import emotion_config as config
7  from pyimagesearch.preprocessing import ImageToArrayPreprocessor
8  from pyimagesearch.callbacks import EpochCheckpoint
9  from pyimagesearch.callbacks import TrainingMonitor
10 from pyimagesearch.io import HDF5DatasetGenerator
11 from pyimagesearch.nn.conv import EmotionVGGNet
12 from keras.preprocessing.image import ImageDataGenerator
13 from keras.optimizers import Adam
14 from keras.models import load_model
15 import keras.backend as K
16 import argparse
17 import os
```

Lines 2 and 3 configure our matplotlib backend so we can save plots to disk. From there, **Lines 6-17** import the remainder of our Python packages. We have used all of these packages before, but I will call your attention to **Line 11** where the newly implemented `EmotionVGGNet` package is imported.

Next, let's parse our command line arguments:

```
19 # construct the argument parse and parse the arguments
20 ap = argparse.ArgumentParser()
```

```
21 ap.add_argument("-c", "--checkpoints", required=True,
22      help="path to output checkpoint directory")
23 ap.add_argument("-m", "--model", type=str,
24      help="path to *specific* model checkpoint to load")
25 ap.add_argument("-s", "--start-epoch", type=int, default=0,
26      help="epoch to restart training at")
27 args = vars(ap.parse_args())
```

These command line arguments are indicative of a `ctrl + c`-style experiment setup. We'll need a `--checkpoints` directory to store EmotionVGGNet weights as the network is trained. If we are loading a specific epoch from disk and (presumably) restarting training, we can supply the `--model` path to the specific checkpoint and provide the `--start-epoch` of the associated checkpoint.

From here, we can instantiate our data augmentation objects:

```
29 # construct the training and testing image generators for data
30 # augmentation, then initialize the image preprocessor
31 trainAug = ImageDataGenerator(rotation_range=10, zoom_range=0.1,
32      horizontal_flip=True, rescale=1 / 255.0, fill_mode="nearest")
33 valAug = ImageDataGenerator(rescale=1 / 255.0)
34 iap = ImageToArrayPreprocessor()
```

As expected, we'll be applying data augmentation to the training set to help reduce overfitting and improve the classification accuracy of our model (**Lines 31 and 32**). But what you may be unfamiliar with is applying data augmentation to the *validation set* (**Line 33**). Why would we need to apply data augmentation to the validation set? Isn't that set supposed to stay static and unchanging?

The answer lies in the `rescale` attribute (which is also part of the training data augmenter). Recall Section 11.1.2 where we converted the `fer2013.csv` file to an HDF5 dataset. We stored these images as raw, unnormalized RGB images, meaning that pixel values are allowed to exist in the range $[0, 255]$. However, it's common practice to either (1) perform mean normalization or (2) scale the pixel intensities down to a more constricted change. Luckily for us, the `ImageDataGenerator` class provided by Keras can automatically perform this scaling for us. We simply need to provide a `rescale` value of $1/255$ – every image will be multiple by this ratio, thus scaling the pixels down to $[0, 1]$.

Let's also initialize our training and validation `HDF5DatasetGenerator` objects:

```
36 # initialize the training and validation dataset generators
37 trainGen = HDF5DatasetGenerator(config.TRAIN_HDF5, config.BATCH_SIZE,
38      aug=trainAug, preprocessors=[iap], classes=config.NUM_CLASSES)
39 valGen = HDF5DatasetGenerator(config.VAL_HDF5, config.BATCH_SIZE,
40      aug=valAug, preprocessors=[iap], classes=config.NUM_CLASSES)
```

The path to the input HDF5 files, batch size, and number of classes are all coming from our `emotion_config`, making it easy for us to change these parameters.

In the case that no specific model checkpoint is supplied, we'll assume we are training our model from the very first epoch:

```
42 # if there is no specific model checkpoint supplied, then initialize
43 # the network and compile the model
```

```
if args["model"] is None:
	print("[INFO] compiling model...")
	model = EmotionVGGNet.build(width=48, height=48, depth=1,
		classes=config.NUM_CLASSES)
	opt = Adam(lr=1e-3)
	model.compile(loss="categorical_crossentropy", optimizer=opt,
		metrics=["accuracy"])
```

Otherwise, we are loading a *specific* checkpoint from disk, updating the learning rate, and restarting training:

```
# otherwise, load the checkpoint from disk
else:
	print("[INFO] loading {}...".format(args["model"]))
	model = load_model(args["model"])

	# update the learning rate
	print("[INFO] old learning rate: {}".format(
		K.get_value(model.optimizer.lr)))
	K.set_value(model.optimizer.lr, 1e-3)
	print("[INFO] new learning rate: {}".format(
		K.get_value(model.optimizer.lr)))
```

Before we can start the training process we must construct our list of `callbacks` used to serialize epoch checkpoints to disk and log accuracy/loss to disk over time:

```
# construct the set of callbacks
figPath = os.path.sep.join([config.OUTPUT_PATH,
	"vggnet_emotion.png"])
jsonPath = os.path.sep.join([config.OUTPUT_PATH,
	"vggnet_emotion.json"])
callbacks = [
	EpochCheckpoint(args["checkpoints"], every=5,
		startAt=args["start_epoch"]),
	TrainingMonitor(figPath, jsonPath=jsonPath,
		startAt=args["start_epoch"])]
```

Our final code block handles training the network:

```
# train the network
model.fit_generator(
	trainGen.generator(),
	steps_per_epoch=trainGen.numImages // config.BATCH_SIZE,
	validation_data=valGen.generator(),
	validation_steps=valGen.numImages // config.BATCH_SIZE,
	epochs=15,
	max_queue_size=config.BATCH_SIZE * 2,
	callbacks=callbacks, verbose=1)

# close the databases
trainGen.close()
valGen.close()
```

Epoch	Learning Rate
$1-20$	$1e-2$
$21-40$	$1e-3$
$41-60$	$1e-4$

Table 11.2: Learning rate schedule used when training EmotionVGGNet on FER2013 in Experiment #1.

The exact number of epochs we'll need to train the network is dependent on the network performance and how we stop training, update the learning rate, and restart training.

In the next section, we'll review four experiments I performed to incrementally improve the classification performance of EmotionVGGNet for facial expression recognition. Use this case study to help you in your own projects – the exact same techniques I use to set baselines, run experiments, investigate the results, and update parameters can be leveraged when you apply deep learning to projects outside of this book.

11.3.1 EmotionVGGNet: Experiment #1

As I always do with my first experiment, I aim to establish a *baseline* that I can incrementally improve upon – your experiment will *rarely* be your best performing model. Keep in mind that obtaining a neural network with high classification is an iterative process. You need to run experiments, gather the results, interpret them, adjust your parameters, and run the experiment again. One more time, this is a *science*, so you need to apply the scientific method – **there are no shortcuts**.

In this experiment, I started with the *SGD optimizer* with a base learning rate of $1e-2$, a momentum term of 0.9, and Nesterov acceleration applied. The (default) Xavier/Glorot initialization method was used to initialize the weights in the `CONV` and `FC` layers. Furthermore, the *only* data augmentation I applied was *horizontal flipping* – no other data augmentation (such as rotation, zoom, etc.) was applied.

I started training using the following command:

```
$ python train_recognizer.py --checkpoints checkpoints
```

And then used the learning rate schedule in Table 11.2 to train the rest of the network:

The loss/accuracy plot for the full 60 epochs can be seen in Figure 11.2. It's interesting to note that as soon as I lowered the learning rate from $1e-2$ to $1e-3$, the network effectively stopped learning. The switch from $1e-3$ to $1e-4$ is practically unnoticeable – with these order of magnitude drops we would expect to see at least *some* rise in accuracy and a corresponding drop in loss.

At the end of the 60th epoch, I examined the validation accuracy and noted that the network was reaching 62.91%. If this were the testing set, we would already be in the top #10 for the Kaggle facial expression recognition leaderboard.

11.3.2 EmotionVGGNet: Experiment #2

Given that SGD led to stagnation in learning when dropping the learning rate, I decided to swap out SGD for Adam, using a base learning rate of $1e-3$. Other than the adjustment to the optimizer, this experiment is identical to the first one. I started training using the following command:

```
$ python train_recognizer.py --checkpoints checkpoints
```

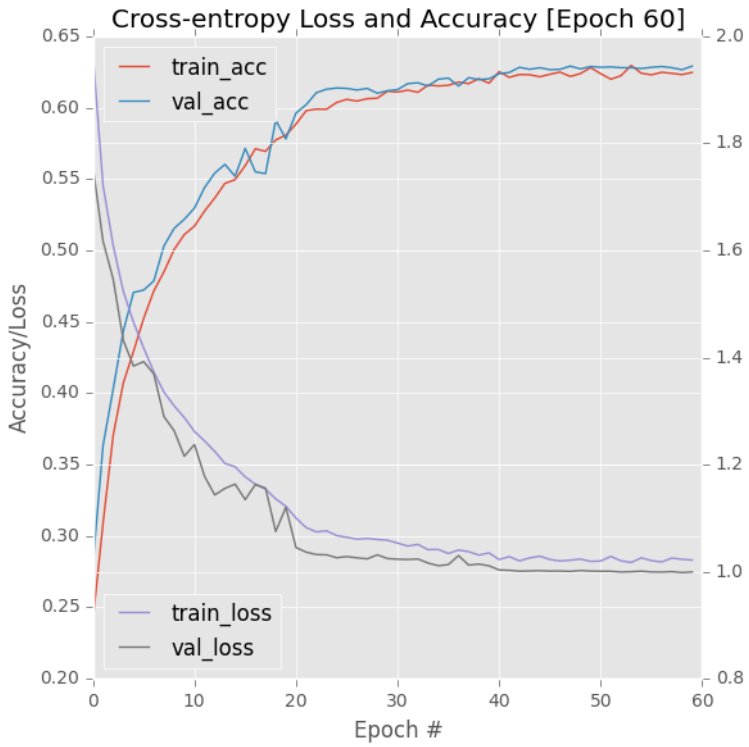


Figure 11.2: Training plot for EmotionVGGNet on FER2013 in Experiment #1. Past epoch 40 learning has stagnated.

At epoch 30 I started to notice sharp divergence between the training loss and validation loss developing (Figure 11.3, *left*), so I stopped training and reduced the learning rate from $1e-3$ to $1e-4$ and then allowed the network to train for another 15 epochs:

```
$ python train_recognizer.py --checkpoints checkpoints \
    --model checkpoints/epoch_30.hdf5 --start-epoch 30
```

However, the result was not good (Figure 11.3, *right*). As we can see, there is clear overfitting – training loss continues to *drop* while validation loss not only *stagnates* but continues to *rise* as well. All that said, the network was still obtaining 66.34% accuracy at the end of the 45th epoch, certainly better than SGD. If I could find a way to curb the overfitting, the Adam optimizer approach would likely perform very well in this situation.

11.3.3 EmotionVGGNet: Experiment #3

A common cure to overfitting is to gather more training data that is representative of your validation and testing set. However, since the FER2013 dataset is pre-compiled *and* we want our results to be relevant in terms of the Kaggle competition, gathering *additional* data is out of the question. Instead, we can apply *data augmentation* to help reduce overfitting.

In my third experiment, I kept my Adam optimizer, but also added in a random rotation range of 10 degrees along with a zoom range of 0.1 (the other augmentation parameters Keras provides did not seem appropriate here). With the new data augmentation scheme in place, I repeated my second experiment:

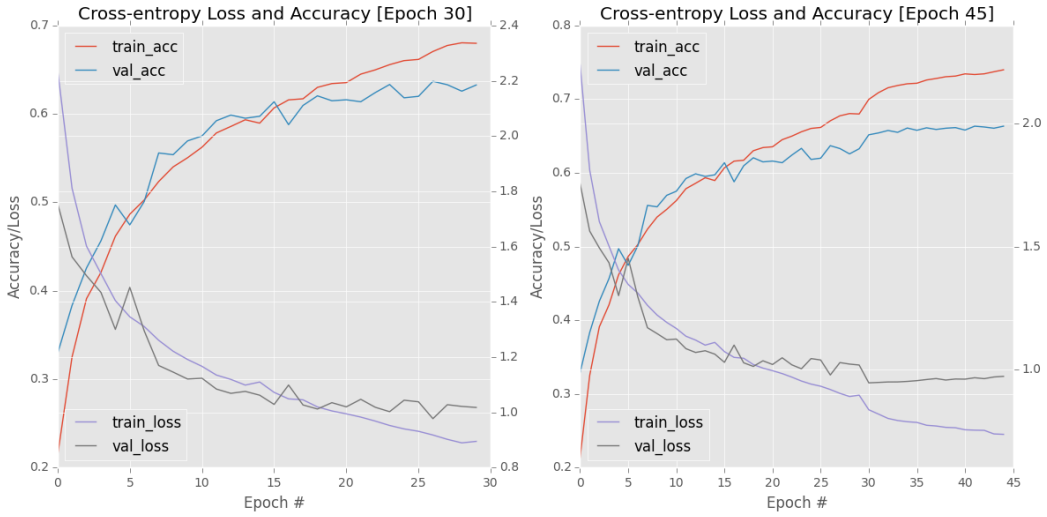


Figure 11.3: **Left:** The first 30 epochs for a $1e-3$ learning rate and the Adam optimizer. **Right:** Lowering the learning rate to $\alpha = 1e-4$ causes clear overfitting.

```
$ python train_recognizer.py --checkpoints checkpoints
```

As the plot below demonstrates, learning is much more stable with saturation starting to occur around epoch 40 (Figure 11.4, *top-left*). At this point, I stopped training, lowered the Adam learning rate from $1e-3$ to $1e-4$, and resumed training:

```
$ python train_recognizer.py --checkpoints checkpoints \
    --model checkpoints/epoch_40.hdf5 --start-epoch 40
```

This process led to a characteristic drop in loss/rise in accuracy which we expect when adjusting the learning rate in this manner (Figure 11.4, *top-right*). However, learning plateaus, so I again stopped training at epoch 60, lowered the learning rate from $1e-4$ to $1e-5$, and resumed training for another 15 epochs:

```
$ python train_recognizer.py --checkpoints checkpoints \
    --model checkpoints/epoch_60.hdf5 --start-epoch 60
```

The final plot of the network can be seen in Figure 11.4 *(bottom)*. As we can see, we aren't at risk of overfitting – the downside is that we aren't seeing any dramatic gains in accuracy past epoch 45. All that said, by applying data augmentation we were able to stabilize learning, reduce overfitting, and allow us to reach 67.53% classification accuracy on the validation set.

11.3.4 EmotionVGGNet: Experiment #4

In my final experiment with FER2013 and EmotionVGGNet I decided to make a few changes:

1. I swapped out Xavier/Glorot initialization (the default used by Keras) for MSRA/He et al. initialization. This swap is due to the fact that He et al. initialization tends to work better for the VGG family of networks (as we've seen in earlier experiments in this book).

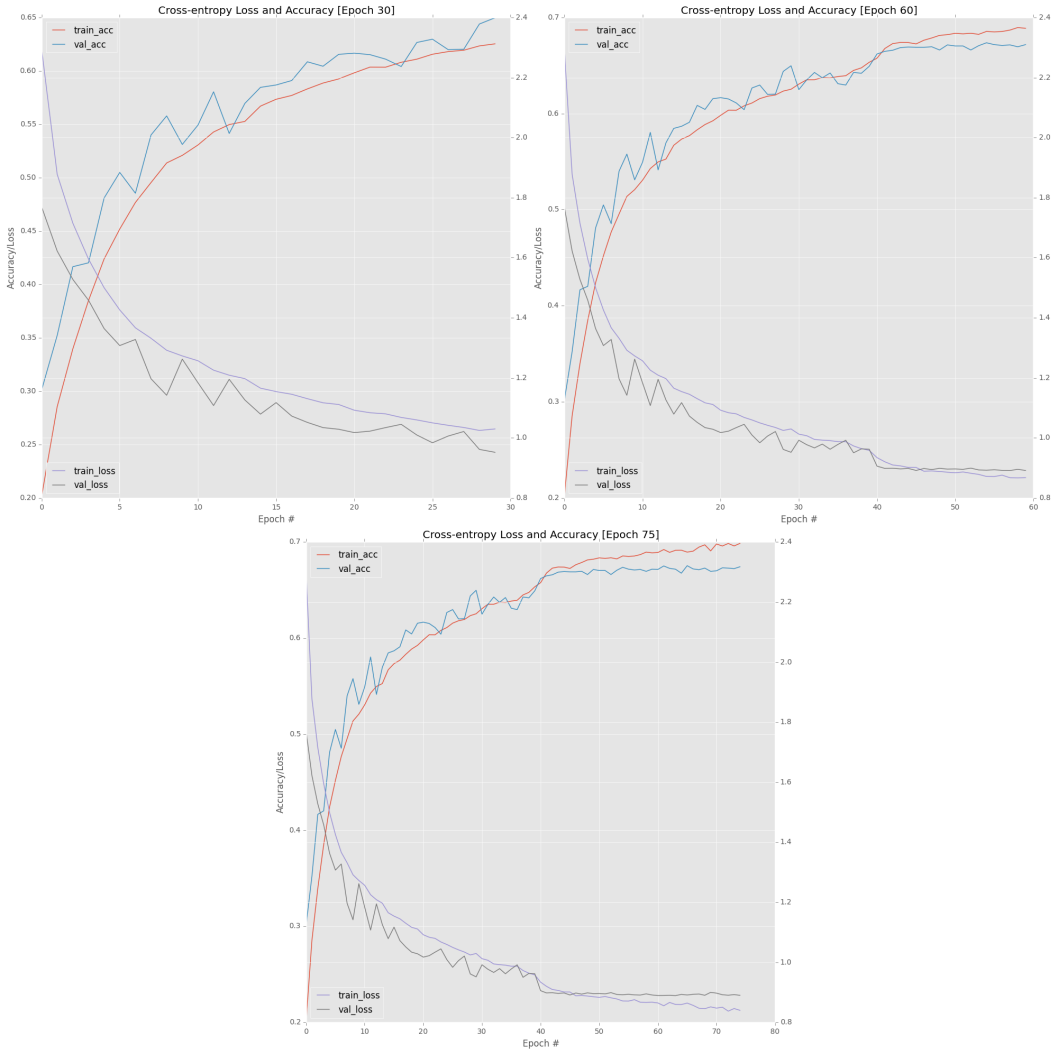


Figure 11.4: **Top-left:** The first 30 epochs for Experiment #3. **Top-right:** Adjusting the learning rate to $1e-4$. **Bottom:** A final adjustment of $\alpha = 1e-5$.

2. I replaced all ReLUs with ELUs in an attempt to further boost accuracy.
3. Given the class imbalance caused by the "disgust" label, I merged "anger" and "disgust" into a single label, per the recommendation of the Mememoji project [33]. The reason for this label merge is two-fold: (1) so we can obtain higher classification accuracy and (2) our model will generalize better when applied to real-time emotion recognition later in this chapter.

In order to merge these two classes, I needed to run `build_dataset.py` again with `NUM_CLASSES` set to six rather than seven.

Again, the Adam optimizer was used with a base learning rate of $1e-3$ which was decayed according to Table 11.3.

The loss/accuracy plot of this experiment can be seen in Figure 11.5. Notice how I always drop the learning rate once validation loss and accuracy plateau. The plot itself looks identical to that of the third experiment; however, when we inspect the output of the 75th epoch we now see EmotionVGGNet is reaching 68.51% accuracy – this is the highest accuracy we have seen thus far. From here, let's move on to evaluating our facial expression recognizer on the test set.

Epoch	Learning Rate
$1-40$	$1e-3$
$40-60$	$1e-4$
$61-75$	$1e-5$

Table 11.3: Learning rate schedule used when training EmotionVGGNet on FER2013 in Experiment #4.

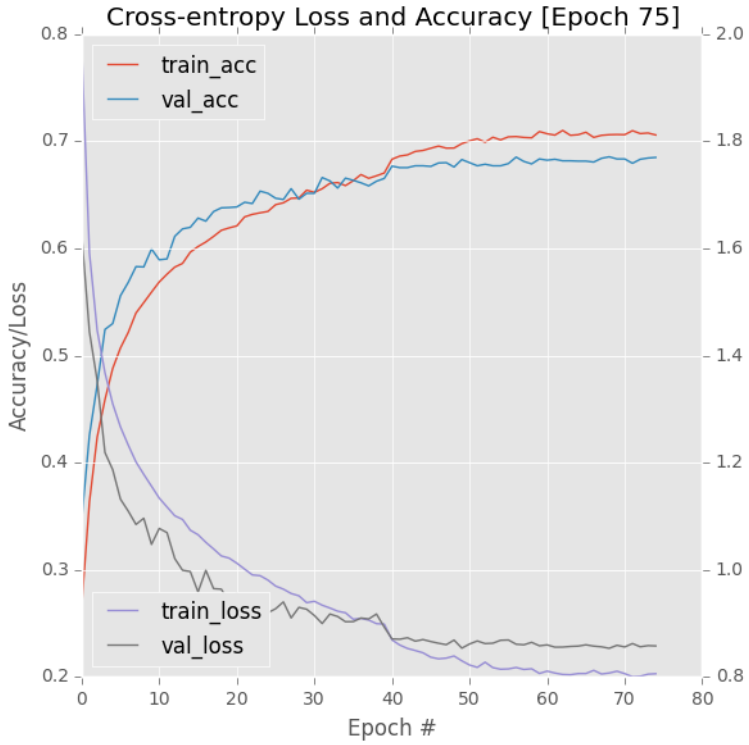


Figure 11.5: Our final experiment training EmotionVGGNet on FER2013. Here we reach 68.51% accuracy by merging the "anger" and "disgust" classes into a single label.

11.4 Evaluating our Facial Expression Recognizer

To evaluate EmotionVGGnet on the FER2013 testing set, let's open up `test_recognizer.py`, and insert the following code:

```
1 # import the necessary packages
2 from config import emotion_config as config
3 from pyimagesearch.preprocessing import ImageToArrayPreprocessor
4 from pyimagesearch.io import HDF5DatasetGenerator
5 from keras.preprocessing.image import ImageDataGenerator
6 from keras.models import load_model
7 import argparse
```

Lines 2-7 import our required Python packages. **Line 2** imports our `emotion_config` so we have access to our project configuration variables. The `HDF5DatasetGenerator` will be needed to

access the testing set (**Line 4**). We'll also import the `ImageDataGenerator` so we can scale the images in FER2013 down to the range $[0, 1]$.

Our script will require only a single command line argument, `--model`, which is the path to the specific model checkpoint to load:

```
9  # construct the argument parse and parse the arguments
10 ap = argparse.ArgumentParser()
11 ap.add_argument("-m", "--model", type=str,
12     help="path to model checkpoint to load")
13 args = vars(ap.parse_args())
```

From there, we initialize our data augmentation class to rescale the images in the testing set:

```
15 # initialize the testing data generator and image preprocessor
16 testAug = ImageDataGenerator(rescale=1 / 255.0)
17 iap = ImageToArrayPreprocessor()
18
19 # initialize the testing dataset generator
20 testGen = HDF5DatasetGenerator(config.TEST_HDF5, config.BATCH_SIZE,
21     aug=testAug, preprocessors=[iap], classes=config.NUM_CLASSES)
```

A pointer to the testing HDF5 dataset is then opened on **Lines 20 and 21**, ensuring that we apply both: (1) our data augmentation for rescaling and (2) our image to Keras-compatible array converter.

Finally, we can load our model checkpoint from disk:

```
23 # load the model from disk
24 print("[INFO] loading {}...".format(args["model"]))
25 model = load_model(args["model"])
```

And evaluate it on the testing set:

```
27 # evaluate the network
28 (loss, acc) = model.evaluate_generator(
29     testGen.generator(),
30     steps=testGen.numImages // config.BATCH_SIZE,
31     max_queue_size=config.BATCH_SIZE * 2)
32 print("[INFO] accuracy: {:.2f}".format(acc * 100))
33
34 # close the testing database
35 testGen.close()
```

To evaluate EmotionVGGNet on FER2013, simply open up a terminal and execute the following command:

```
$ python test_recognizer.py --model checkpoints/epoch_75.hdf5
[INFO] loading checkpoints/epoch_75.hdf5...
[INFO] accuracy: 66.96
```

As my results demonstrate, we were able to obtain **66.96**% accuracy on the test set, enough for us to claim the 5th position on the Kaggle Facial Expression Recognition Challenge [31].

This 66.96% classification result was obtained from the *6-class* variant of FER2013, not the *7-class* original version in the Kaggle recognition challenge. That said, we can easily re-train the network on the 7-class version and obtain a similar accuracy. The reason we are ending with the 6-class network is so we can obtain more meaningful results when applied to real-time emotion detection in the next section.

11.5 Emotion Detection in Real-time

Now that our CNN is trained and evaluated, let's apply it to detect emotion and facial expressions in video streams in real-time. Open up a new file, name it `emotion_detector.py`, and insert the following code:

```
1  # import the necessary packages
2  from keras.preprocessing.image import img_to_array
3  from keras.models import load_model
4  import numpy as np
5  import argparse
6  import imutils
7  import cv2
8
9  # construct the argument parse and parse the arguments
10 ap = argparse.ArgumentParser()
11 ap.add_argument("-c", "--cascade", required=True,
12      help="path to where the face cascade resides")
13 ap.add_argument("-m", "--model", required=True,
14      help="path to pre-trained emotion detector CNN")
15 ap.add_argument("-v", "--video",
16      help="path to the (optional) video file")
17 args = vars(ap.parse_args())
```

Lines 2-7 import our required Python packages. From there, we parse our command line arguments. Our script requires two switches, followed by a third optional one. The `--cascade` switch is the path to our face detection Haar cascade (included in the downloads portion of this book). The `--model` is our pre-trained CNN that we will use to detect emotion. Finally, if you wish to apply emotion detection to a *video file* rather than a *video stream*, you can supply the path to the file via the `--video` switch.

From there, let's load our face detection cascade, emotion detection CNN, as well as initialize the list of emotion labels that our CNN can predict:

```
19 # load the face detector cascade, emotion detection CNN, then define
20 # the list of emotion labels
21 detector = cv2.CascadeClassifier(args["cascade"])
22 model = load_model(args["model"])
23 EMOTIONS = ["angry", "scared", "happy", "sad", "surprised",
24      "neutral"]
```

Notice how our `EMOTIONS` list contains six entries, implying that we are using the 6-class version of FER2013 for better accuracy.

Our following code block instantiates a `cv2.VideoCapture` object based on whether we are (1) accessing a webcam or (2) reading from a video file:

```
26 # if a video path was not supplied, grab the reference to the webcam
27 if not args.get("video", False):
28     camera = cv2.VideoCapture(1)
29
30 # otherwise, load the video
31 else:
32     camera = cv2.VideoCapture(args["video"])
```

We are now ready to start looping over frames from the video pointer:

```
34 # keep looping
35 while True:
36     # grab the current frame
37     (grabbed, frame) = camera.read()
38
39     # if we are viewing a video and we did not grab a
40     # frame, then we have reached the end of the video
41     if args.get("video") and not grabbed:
42         break
```

Line 37 reads the next `frame` from the video stream. If the frame was not `grabbed` (i.e., set to `False`) *and* we are reading frames from a video stream, we have reached the end of the file, so we should break from the loop (**Lines 41 and 42**).

Otherwise, it's time to pre-process the frame by resizing it to have a width of 300 pixels and converting it to grayscale:

```
44     # resize the frame and convert it to grayscale
45     frame = imutils.resize(frame, width=300)
46     gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
47
48     # initialize the canvas for the visualization, then clone
49     # the frame so we can draw on it
50     canvas = np.zeros((220, 300, 3), dtype="uint8")
51     frameClone = frame.copy()
52
53     # detect faces in the input frame, then clone the frame so that
54     # we can draw on it
55     rects = detector.detectMultiScale(gray, scaleFactor=1.1,
56         minNeighbors=5, minSize=(30, 30),
57         flags=cv2.CASCADE_SCALE_IMAGE)
```

We initialize an empty NumPy `canvas` (**Line 50**) which has a width of 300px and a height of 200px. We'll be using the `canvas` to draw the *probability distribution* predicted by our CNN, which will enable us to visualize the range and mixture of the emotions.

Lines 55-57 then detect faces in the `frame` using OpenCV's pre-trained Haar cascade. If you are interested in learning more about object detection via Haar cascades, please refer to *Practical Python and OpenCV* [34] (http://pyimg.co/ppao) and the PyImageSearch Gurus course [35] (http://pyimg.co/gurus).

In the next code block, we prepare the face ROI for classification via the CNN:

```
59      # ensure at least one face was found before continuing
60      if len(rects) > 0:
61          # determine the largest face area
62          rect = sorted(rects, reverse=True,
63              key=lambda x: (x[2] - x[0]) * (x[3] - x[1]))[0]
64          (fX, fY, fW, fH) = rect
65
66          # extract the face ROI from the image, then pre-process
67          # it for the network
68          roi = gray[fY:fY + fH, fX:fX + fW]
69          roi = cv2.resize(roi, (48, 48))
70          roi = roi.astype("float") / 255.0
71          roi = img_to_array(roi)
72          roi = np.expand_dims(roi, axis=0)
```

Line 60 ensures that *at least* one face was detected in the `frame`. Provided there was at least one face detected, we sort the bounding box list `rect` according to the *size* of the bounding box, with large faces at the front of the list (**Lines 62 and 64**).

We could certainly apply emotion detection and facial expression recognition to *every* face in the frame; however, the point of this script is to demonstrate how we can (1) detect the most dominant facial expression and (2) plot a *distribution* of emotions. Plotting this distribution for *every* face in the frame would be distracting and harder for us to visualize. Thus, I will leave it as an exercise to the reader to loop over each of the `rects` individually – as a matter of simplicity, we'll simply be using the *largest* face region in our example.

Line 68 extracts the face region from the grayscale image via NumPy array slicing. We then pre-process the `roi` by resizing it to a fixed 48×48 pixels (the input size required by EmotionVGGNet). From there, we convert the `roi` to a floating point data type, scale it to the range $[0, 1]$, and convert it to a Keras-compatible array (**Lines 69-72**).

Now that the `roi` has been pre-processed, we can pass it through our `model` to obtain the class probabilities:

```
74          # make a prediction on the ROI, then lookup the class
75          # label
76          preds = model.predict(roi)[0]
77          label = EMOTIONS[preds.argmax()]
78
79          # loop over the labels + probabilities and draw them
80          for (i, (emotion, prob)) in enumerate(zip(EMOTIONS, preds)):
81              # construct the label text
82              text = "{}: {:.2f}%".format(emotion, prob * 100)
83
84              # draw the label + probability bar on the canvas
85              w = int(prob * 300)
86              cv2.rectangle(canvas, (5, (i * 35) + 5),
87                  (w, (i * 35) + 35), (0, 0, 255), -1)
88              cv2.putText(canvas, text, (10, (i * 35) + 23),
89                  cv2.FONT_HERSHEY_SIMPLEX, 0.45,
90                  (255, 255, 255), 2)
```

Line 76 makes a call to the `predict` method of `model` which returns the predicted class label probabilities. The `label` is thus the label with the largest associated probability (**Line 77**).

However, since human facial expressions are often a *mixture* of emotions, it's much more interesting to examine the probability distribution of the labels. To accomplish this determination, we loop over the labels and associated probabilities on **Line 80**. **Line 86 and 87** draw a bar chart where each bar width is proportional to the predicted class label probability. **Lines 88-90** then draw the name of the `label` on the `canvas`.

If you are interested in understanding how these OpenCV drawing functions work in more detail, please refer to *Practical Python and OpenCV* [34] (http://pyimg.co/ppao).

Our next code block handles drawing the `label` with the highest probability to our screen as well as drawing a bound box surrounding the face we predicted emotions for:

```
92            # draw the label on the frame
93            cv2.putText(frameClone, label, (fX, fY - 10),
94                 cv2.FONT_HERSHEY_SIMPLEX, 0.45, (0, 0, 255), 2)
95            cv2.rectangle(frameClone, (fX, fY), (fX + fW, fY + fH),
96                 (0, 0, 255), 2)
```

The last code block in this example simply displays the output images to our screen so we can visualize the results in real-time:

```
98       # show our classifications + probabilities
99       cv2.imshow("Face", frameClone)
100      cv2.imshow("Probabilities", canvas)
101
102      # if the 'q' key is pressed, stop the loop
103      if cv2.waitKey(1) & 0xFF == ord("q"):
104           break
105
106 # cleanup the camera and close any open windows
107 camera.release()
108 cv2.destroyAllWindows()
```

If you wish to apply the emotion detector to a webcam video stream, open up your terminal and execute the following command:

```
$ python emotion_detector.py --cascade haarcascade_frontalface_default.xml \
     --model checkpoints/epoch_75.hdf5
```

Otherwise, if you instead are applying facial expression prediction to a video file, update the command to use the `--video` switch to point to your video file residing on disk:

```
$ python emotion_detector.py --cascade haarcascade_frontalface_default.xml \
     --model checkpoints/epoch_75.hdf5 --video path/to/your/video.mp4
```

In the chapter downloads associated with this book, I have provided an example video that you can use to verify that your emotion predictor is working correctly. For example, in the following frame my face is clearly happy (Figure 11.6, *top-left*). While in the *top-right* my facial expression has changed to angry. However, notice that the probability of sad is also higher, implying that

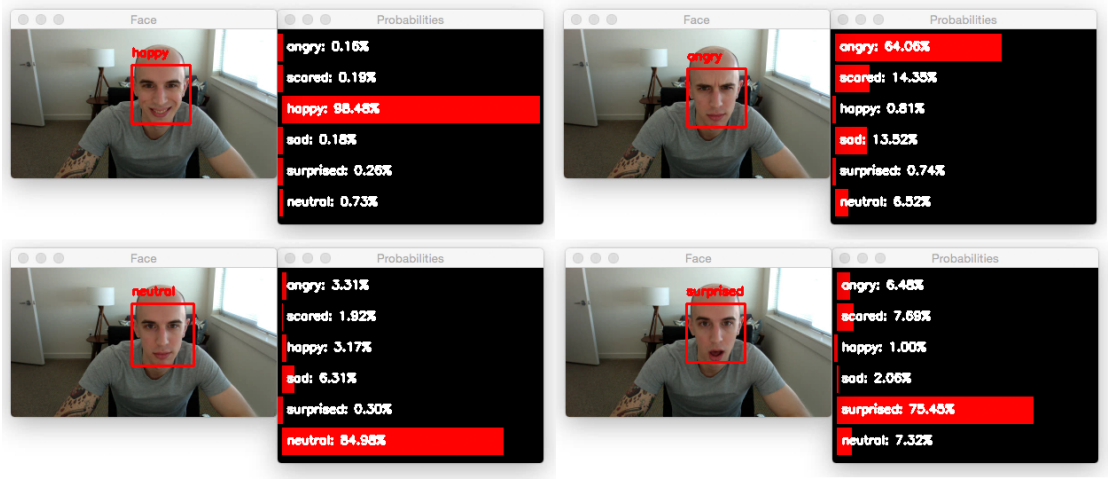


Figure 11.6: Examples of facial expression recognition using our Convolutional Neural Network. In each case we are able to correctly recognize the emotion. Furthermore, notice that the probability distribution represents facial expression as a *mixture* of emotions.

there is a mixture of emotions. A few seconds later, my face returns to a resting, neutral position (*bottom-left*). Finally, in *bottom-right*, I am clearly surprised.

It's important to note that this model was *not* trained on any example images of myself – the *only* dataset used was FER2013. Given that our model is able to *correctly predict* my facial expressions, we can say that our EmotionVGGNet expresses a very good ability to generalize.

11.6 Summary

In this chapter, we learned how to implement a Convolutional Neural Network capable of predicting emotion and facial expressions. To accomplish this task, we trained a VGG-like CNN named EmotionVGGNet. This network consisted of two `CONV` layers stacked on top of each other with the number of filters doubling in each block. It was important that our CNN be:

1. Deep enough to obtain high accuracy.
2. But not so deep that it would be impossible to run in real-time on the CPU.

We then trained our CNN on the FER2013 dataset, part of the Kaggle Emotion and Facial Expression Recognition challenge. Overall, we were able to obtain **66.96**% accuracy, enough to claim the #5 position on the leaderboard. Further accuracy can likely be obtained by being more aggressive with our data augmentation, deepening the network, increasing the number of layers, and adding in regularization.

Finally, we ended this chapter by creating a Python script that can (1) detect faces in a video stream and (2) apply our pre-trained CNN to recognize dominant facial expression in real-time. Furthermore, we also included the *probability distribution* for each emotion, enabling us to more easily interpret the results of our network.

Again, keep in mind that as humans, our emotions are *fluid* and are constantly changing. Furthermore, our emotions are often *mixtures* of each other – we are rarely, if ever, 100% happy or 100% sad – instead we are always some blend of feelings. Because of this fact, it's important that you examine the probability distribution returned by EmotionVGGNet when trying to label the facial expression of a given person.

12. Case Study: Correcting Image Orientation

In this case study, we are going to learn how to apply transfer learning (specifically feature extraction) to automatically *detect* and *correct* the orientation of an image. The reason we are using feature extraction from a pre-trained Convolutional Neural Network is two-fold:

1. To demonstrate that the filters learned by a CNN trained on ImageNet are *not* rotation invariant across a full 360-degree spectrum (otherwise, the features could not be used to discriminate between image rotation).
2. Transfer learning via feature extraction obtains the highest accuracy when predicting image orientation.

As I mentioned in Chapter 11 of the *Starter Bundle*, it's a common misconception that the *individual filters* learned by a CNN are invariant to rotation – *the filters themselves are not*. Instead, the CNN is able to *learn a set of filters* that activate when they see a particular object under a given rotation. Therefore, the filters themselves are not rotation invariant, but the CNN is able to obtain some level of rotation invariance due to the number of filters it has learned, with ideally some of these filters activating under various rotations. As we'll see in this chapter, CNNs are not fully rotation invariant, otherwise, it would be impossible for us to determine the orientation of an image strictly from the features extracted during the fine-tuning phase.

12.1 The Indoor CVPR Dataset

The dataset we'll be using for this case study is the Indoor Scene Recognition (also called Indoor CVPR) dataset [36] released by MIT. This database contains 67 indoor categories room/scene categories, including homes, offices, public spaces, stores, and many more. A sample of this dataset can be seen in Figure 12.1.

The reason we are using this dataset is because it is very easy for a human to determine if a natural scene image is incorrectly oriented – our goal is to replicate this level of performance using a Convolutional Neural Network. However, all images in Indoor CVPR are correctly oriented; therefore, we need to build our own dataset from Indoor CVPR with labeled images under various rotations.

To download the Indoor CVPR dataset use this link:

Figure 12.1: A sample of the classes and images from the Indoor Scene Recognition dataset [36].

http://pyimg.co/x772x

And form there click the "Download" link. The entire .tar archive is $\approx 2.4GB$ so plan your download accordingly. Once the file has downloaded, unarchive it. Once unarchived, you'll find a directory named `Images` which contains a number of subdirectories, each one containing a particular class label in the dataset:

```
$ cd Images
$ ls -l | head -n 10
total 0
drwxr-xr-x@ 610 adrianrosebrock  staff  20740 Mar 16  2009 airport_inside
drwxr-xr-x@ 142 adrianrosebrock  staff   4828 Mar 16  2009 artstudio
drwxr-xr-x@ 178 adrianrosebrock  staff   6052 Mar 16  2009 auditorium
drwxr-xr-x@ 407 adrianrosebrock  staff  13838 Mar 16  2009 bakery
drwxr-xr-x@ 606 adrianrosebrock  staff  20604 Mar 16  2009 bar
drwxr-xr-x@ 199 adrianrosebrock  staff   6766 Mar 16  2009 bathroom
drwxr-xr-x@ 664 adrianrosebrock  staff  22576 Mar 16  2009 bedroom
drwxr-xr-x@ 382 adrianrosebrock  staff  12988 Mar 16  2009 bookstore
drwxr-xr-x@ 215 adrianrosebrock  staff   7310 Mar 16  2009 bowling
```

To keep the Indoor CVPR dataset organized I created a new directory named `indoor_cvpr`, moved the `Images` directory inside of (changing the uppercase "I" to lowercase "i"), and creating two new subdirectories – `hdf5` and `rotated_images`. My directory structure for this project is as follows:

```
--- indoor_cvpr
|    |--- hdf5
|    |--- images
|    |--- rotated_images
```

The `hdf5` directory will store the features extracted from our input `images` using a pre-trained Convolutional Neural Network. In order to generate our training data, we will create a custom Python script that generates randomly rotates images. These rotated images will be stored in `rotated_images`. The features extracted from these images will be stored in HDF5 datasets.

12.1.1 Building the Dataset

Before we get started let's take a look at our project structure:

```
--- image_orientation
|    |--- creat_dataset.py
```

```
|    |--- extract_features.py
|    |--- indoor_cvpr/
|    |--- models/
|    |--- orient_images.py
|    |--- train_model.py
```

We'll be using the `create_dataset.py` script to build the training and testing sets for our input dataset. From there `extract_features.py` will be used to create an HDF5 file for the dataset splits. Given the extracted features we can use `train_model.py` to train a Logistic Regression classifier to recognize image orientations and save the resulting model in the `models` directory. Finally, `orient_images.py` can be applied to orient testing input images.

Just as I have done with the ImageNet dataset, I created a sym-link to `indoor_cvpr` to make it easier to type out commands; however, you can ignore this step if you wish and instead specify the full path to the input/output directories when executing the Python scripts.

Let's go ahead and learn how we can construct our own custom image orientation dataset from an *existing* dataset. Make sure you have downloaded the `.tar` file from the Indoor Scene Recognition dataset, and from there, open up a new file, name it `create_dataset.py`, and we'll get it work:

```
1  # import the necessary packages
2  from imutils import paths
3  import numpy as np
4  import progressbar
5  import argparse
6  import imutils
7  import random
8  import cv2
9  import os
10
11 # construct the argument parse and parse the arguments
12 ap = argparse.ArgumentParser()
13 ap.add_argument("-d", "--dataset", required=True,
14      help="path to input directory of images")
15 ap.add_argument("-o", "--output", required=True,
16      help="path to output directory of rotated iamges")
17 args = vars(ap.parse_args())
```

Lines 2-9 import our required Python packages, all of which we have reviewed earlier in this book. We'll again be using the optional `progressbar` package to display updates to our terminal as our dataset creation process runs. If you do not want to use `progressbar`, simply comment out the few lines in this script that reference it.

Lines 12-17 then parse our command line arguments. We only need two command line arguments here, `--dataset`, the path to the directory containing our input images (the Indoor CVPR dataset), along with `--output` the path to the directory where our labeled, rotated images will be stored.

Next, let's randomly sample 10,000 images from our `--dataset` directory and initialize our progress bar:

```
19 # grab the paths to the input images (limiting ourselves to 10,000
20 # images) and shuffle them to make creating a training and testing
21 # split easier
```

```
22 imagePaths = list(paths.list_images(args["dataset"]))[:10000]
23 random.shuffle(imagePaths)
24
25 # initialize a dictionary to keep track of the number of each angle
26 # chosen so far, then initialize the progress bar
27 angles = {}
28 widgets = ["Building Dataset: ", progressbar.Percentage(), " ",
29      progressbar.Bar(), " ", progressbar.ETA()]
30 pbar = progressbar.ProgressBar(maxval=len(imagePaths),
31      widgets=widgets).start()
```

Given our sampled `imagePaths`, we are now ready to create our rotated image dataset:

```
33 # loop over the image paths
34 for (i, imagePath) in enumerate(imagePaths):
35      # determine the rotation angle, and load the image
36      angle = np.random.choice([0, 90, 180, 270])
37      image = cv2.imread(imagePath)
38
39      # if the image is None (meaning there was an issue loading the
40      # image from disk, simply skip it)
41      if image is None:
42           continue
```

On **Line 34** we start looping over each of the individual sampled images. **Line 36** then randomly selects the rotation angle in which we are going to rotate the image – either by 0 degrees (no change), 90 degrees, 180 degrees (flipped vertically), or 270 degrees. **Line 37** loads our `image` from disk. I've found that some images in the Indoor CVPR dataset do not load properly from disk due to JPEG encoding issues; therefore, **Lines 41 and 42** ensure the image is loaded properly, and if not, we simply skip the image.

The next code block will handle rotating our `image` by the randomly selected `angle` and then writing the image to disk:

```
44      # rotate the image based on the selected angle, then construct
45      # the path to the base output directory
46      image = imutils.rotate_bound(image, angle)
47      base = os.path.sep.join([args["output"], str(angle)])
48
49      # if the base path does not exist already, create it
50      if not os.path.exists(base):
51           os.makedirs(base)
52
53      # extract the image file extension, then construct the full path
54      # to the output file
55      ext = imagePath[imagePath.rfind("."):]
56      outputPath = [base, "image_{}{}".format(
57           str(angles.get(angle, 0)).zfill(5), ext)]
58      outputPath = os.path.sep.join(outputPath)
59
60      # save the image
61      cv2.imwrite(outputPath, image)
```

Line 46 rotates our `image` by the `angle`, ensuring the *entire image* stays in the field of view (http://pyimg.co/7xnk6). We then determine the base output path for the image, again, based on the `angle`. This implies that our output dataset will have the following directory structure:

```
/output/{angle_name}/{image_filename}.jpg
```

Lines 50 and 51 check to see if the `/output/{angle_name}` directory structure exists, and if not, creates the required sub-directory. Given the base output path, we can derive the path to the actual image on **Lines 55-58** by extracting the:

1. Original image file extension.
2. The number of images currently selected with the given `angle`.

Finally, **Line 61** writes our rotated `image` to disk in the correct angle sub-directory, ensuring we can easily determine the label of the image simply by examining the file path. For example, we know the output image path `/rotated_images/180/image_00000.jpg` has been rotated by 180 degrees because it resides in the `180` sub-directory.

In order to keep track of the number of images per angle, we update the `angles` bookkeeping dictionary in the following code block:

```
63      # update the count for the angle
64      c = angles.get(angle, 0)
65      angles[angle] = c + 1
66      pbar.update(i)
```

Finally, we display some statistics on the number of images per angle in our dataset:

```
68  # finish the progress bar
69  pbar.finish()
70
71  # loop over the angles and display counts for each of them
72  for angle in sorted(angles.keys()):
73      print("[INFO] angle={}: {:,}".format(angle, angles[angle]))
```

To build our rotated image dataset, open up a terminal and execute the following command:

```
$ python create_dataset.py --dataset indoor_cvpr/images \
    --output indoor_cvpr/rotated_images
Building Dataset: 100% |####################################| Time: 0:01:19
[INFO] angle=0: 2,487
[INFO] angle=90: 2,480
[INFO] angle=180: 2,525
[INFO] angle=270: 2,483
```

As you can see from the output, the entire dataset creation process took 1m19s with $\approx 2,500$ images per rotation angle, give or take a few images due to the random sampling process. Now that we have created our dataset, we can move on to applying transfer learning via feature extraction – these features will then be used in a Logistic Regression classifier to predict (and correct) the orientation of an input image.

12.2 Extracting Features

To extract features from our dataset, we'll be using the VGG16 network architecture that has been pre-trained on the ImageNet dataset. This script is *identical* to the one we implemented in Chapter 3 of the *Practitioner Bundle*. The reason we are able to reuse the same code is due to:

1. The modularity of the tools we built throughout this entire book, ensuring we can reuse scripts provided our input datasets follow a *specific* directory structure.
2. Ensuring our images follow the directory pattern `/dataset_name/class_label/example_image.jpg`

As a matter of completeness, I'll review `extract_features.py` below, but for more details, please refer to Chapter 3 of the *Practitioner Bundle*:

```
1  # import the necessary packages
2  from keras.applications import VGG16
3  from keras.applications import imagenet_utils
4  from keras.preprocessing.image import img_to_array
5  from keras.preprocessing.image import load_img
6  from sklearn.preprocessing import LabelEncoder
7  from pyimagesearch.io import HDF5DatasetWriter
8  from imutils import paths
9  import numpy as np
10 import progressbar
11 import argparse
12 import random
13 import os
```

Lines 2-13 import our required Python packages. **Line 2** imports the VGG16 network architecture that we'll be treating as a feature extractor. Features extracted by the CNN will be written to an HDF5 dataset using our `HDF5DatasetWriter`.

Next, let's parse our command line arguments:

```
15 # construct the argument parse and parse the arguments
16 ap = argparse.ArgumentParser()
17 ap.add_argument("-d", "--dataset", required=True,
18      help="path to input dataset")
19 ap.add_argument("-o", "--output", required=True,
20      help="path to output HDF5 file")
21 ap.add_argument("-b", "--batch-size", type=int, default=32,
22      help="batch size of images to be passed through network")
23 ap.add_argument("-s", "--buffer-size", type=int, default=1000,
24      help="size of feature extraction buffer")
25 args = vars(ap.parse_args())
```

Here we need to supply an input path to our `--dataset` of rotated images on disk along with the `--output` path to our HDF5 file. Given the path to our `--dataset` we can grab the paths to the individual images:

```
27 # store the batch size in a convenience variable
28 bs = args["batch_size"]
29
30 # grab the list of images that we'll be describing then randomly
31 # shuffle them to allow for easy training and testing splits via
```

```
# array slicing during training time
print("[INFO] loading images...")
imagePaths = list(paths.list_images(args["dataset"]))
random.shuffle(imagePaths)
```

Followed by encoding the labels by extracting the orientation angle from the image path:

```
# extract the class labels from the image paths then encode the
# labels
labels = [p.split(os.path.sep)[-2] for p in imagePaths]
le = LabelEncoder()
labels = le.fit_transform(labels)
```

The following code block handles loading our pre-trained VGG16 network from disk, ensuring the FC layers are left (allowing us to perform feature extraction):

```
# load the VGG16 network
print("[INFO] loading network...")
model = VGG16(weights="imagenet", include_top=False)

# initialize the HDF5 dataset writer, then store the class label
# names in the dataset
dataset = HDF5DatasetWriter((len(imagePaths), 512 * 7 * 7),
     args["output"], dataKey="features", bufSize=args["buffer_size"])
dataset.storeClassLabels(le.classes_)
```

We also initialize our HDF5DatasetWriter to write the extracted features to disk. And I'll initialize a progress bar so we can keep track of the feature extraction process:

```
# initialize the progress bar
widgets = ["Extracting Features: ", progressbar.Percentage(), " ",
     progressbar.Bar(), " ", progressbar.ETA()]
pbar = progressbar.ProgressBar(maxval=len(imagePaths),
     widgets=widgets).start()
```

We are now ready to apply transfer learning via feature extraction:

```
# loop over the images in patches
for i in np.arange(0, len(imagePaths), bs):
     # extract the batch of images and labels, then initialize the
     # list of actual images that will be passed through the network
     # for feature extraction
     batchPaths = imagePaths[i:i + bs]
     batchLabels = labels[i:i + bs]
     batchImages = []

     # loop over the images and labels in the current batch
     for (j, imagePath) in enumerate(batchPaths):
          # load the input image using the Keras helper utility
          # while ensuring the image is resized to 224x224 pixels
          image = load_img(imagePath, target_size=(224, 224))
```

```
73            image = img_to_array(image)
74
75            # preprocess the image by (1) expanding the dimensions and
76            # (2) subtracting the mean RGB pixel intensity from the
77            # ImageNet dataset
78            image = np.expand_dims(image, axis=0)
79            image = imagenet_utils.preprocess_input(image)
80
81            # add the image to the batch
82            batchImages.append(image)
```

Line 60 starts looping over all images in our `imagePaths` list in batches. For each of the images, we load them from disk, pre-process them, and store them in a `batchImages` list. The `batchImages` are then passed through the pre-trained VGG16 network yielding us our features which are written to the HDF5 dataset:

```
84        # pass the images through the network and use the outputs as
85        # our actual features
86        batchImages = np.vstack(batchImages)
87        features = model.predict(batchImages, batch_size=bs)
88
89        # reshape the features so that each image is represented by
90        # a flattened feature vector of the 'MaxPooling2D' outputs
91        features = features.reshape((features.shape[0], 512 * 7 * 7))
92
93        # add the features and labels to our HDF5 dataset
94        dataset.add(features, batchLabels)
95        pbar.update(i)
96
97    # close the dataset
98    dataset.close()
99    pbar.finish()
```

To extract features from our rotated images dataset, simply execute the following command:

```
$ python extract_features.py --dataset indoor_cvpr/rotated_images \
      --output indoor_cvpr/hdf5/orientation_features.hdf5
[INFO] loading images...
[INFO] loading network...
Extracting Features: 100% |#####################################| Time: 0:02:13
```

Depending on whether you are using your CPU or GPU, this process may take a few minutes to a few hours. After the process completes, you'll find a file named `orientation_features.hdf5` in your `output` directory:

```
$ ls -l indoor_cvpr/hdf5/
total 1955192
-rw-rw-r-- 1 adrian adrian 2002108504 Nov 21  2016 orientation_features.hdf5
```

This file contains the features extracted by our CNN on top of which which we'll train Logistic Regression classifier. Again, for more information on transfer learning via feature extraction, please see Chapter 3 of the *Practitioner Bundle*.

12.3 Training an Orientation Correction Classifier

Training a classifier to predict image orientation will be accomplished using the `train_model.py` script from the *Practitioner Bundle* as well – we simply need to supply the path to our input HDF5 dataset and our script will take care of tuning the Logistic Regression hyperparameters and writing the output model to disk. Again, as a matter of completeness, I will review `train_model.py` below, but you'll want to refer to Chapter 3 of the *Practitioner Bundle* for more detailed information on this script.

```
1  # import the necessary packages
2  from sklearn.linear_model import LogisticRegression
3  from sklearn.model_selection import GridSearchCV
4  from sklearn.metrics import classification_report
5  import argparse
6  import pickle
7  import h5py
8
9  # construct the argument parse and parse the arguments
10 ap = argparse.ArgumentParser()
11 ap.add_argument("-d", "--db", required=True,
12      help="path HDF5 database")
13 ap.add_argument("-m", "--model", required=True,
14      help="path to output model")
15 ap.add_argument("-j", "--jobs", type=int, default=-1,
16      help="# of jobs to run when tuning hyperparameters")
17 args = vars(ap.parse_args())
```

Lines 2-7 import our required Python packages while **Lines 10-17** parse our command line arguments. We only need to supply two command line arguments here, `--db`, which is the path to our input HDF5 dataset, and `--model`, the path to our output serialized Logistic Regression classifier once it has been trained.

We then construct a training and testing split based on the number of entries in the database, using 75% of the data for testing and 25% for testing:

```
19 # open the HDF5 database for reading then determine the index of
20 # the training and testing split, provided that this data was
21 # already shuffled *prior* to writing it to disk
22 db = h5py.File(args["db"], "r")
23 i = int(db["labels"].shape[0] * 0.75)
```

To tune the parameters of our Logistic Regression classifier we'll perform a grid search below:

```
25 # define the set of parameters that we want to tune then start a
26 # grid search where we evaluate our model for each value of C
27 print("[INFO] tuning hyperparameters...")
28 params = {"C": [0.01, 0.1, 1.0, 10.0, 100.0, 1000.0, 10000.0]}
29 model = GridSearchCV(LogisticRegression(), params, cv=3,
30      n_jobs=args["jobs"])
31 model.fit(db["features"][:i], db["labels"][:i])
32 print("[INFO] best hyperparameters: {}".format(model.best_params_))
```

Based on the best hyperparameters we'll evaluate the model on the testing data:

```
# evaluate the model
print("[INFO] evaluating...")
preds = model.predict(db["features"][i:])
print(classification_report(db["labels"][i:], preds,
     target_names=db["label_names"]))
```

And then finally write the trained classifier to disk:

```
# serialize the model to disk
print("[INFO] saving model...")
f = open(args["model"], "wb")
f.write(pickle.dumps(model.best_estimator_))
f.close()

# close the database
db.close()
```

To train our Logistic Regression classifier on the features extracted from our VGG16 network, just execute the following command:

```
$ python train_model.py --db indoor_cvpr/hdf5/orientation_features.hdf5 \
     --model models/orientation.cpickle
[INFO] tuning hyperparameters...
[INFO] best hyperparameters: {'C': 0.01}
[INFO] evaluating...
             precision    recall  f1-score   support

          0       0.95      0.93      0.94       646
        180       0.92      0.94      0.93       612
        270       0.92      0.92      0.92       635
         90       0.93      0.94      0.93       601

avg / total       0.93      0.93      0.93      2494
```

Once `train_model.py` finishes executing, you'll notice that our classifier obtains 93% accuracy, implying that 93% of the time our model can correctly predict the orientation of the image. Given that we can *predict* the orientation of the image we can then *correct it* in the following section.

Again, it's worth mentioning here that we would not be able to obtain such a high prediction accuracy if VGG16 was *truly* rotation invariant. Instead, the (collective) filters inside VGG16 are able to recognize objects in various rotations – the individual filters themselves *are not* rotation invariant. If these filters truly were rotation invariant, then it would be impossible for us to determine image orientation due the fact that the extracted features would be near identical regardless of image orientation.

This is an important lesson to learn when you are studying Convolutional Neural Networks and expect to deploy them to real-world situations. If you intend on recognizing objects under *any* orientation, you'll want to ensure that your training data sufficiently reflects that, or use a feature extraction + machine learning method that naturally handles rotation.

12.4 Correcting Orientation

To demonstrate that our VGG16 feature extraction + Logistic Regression classifier can sufficiently classify images, let's write a Python script that applies the pipeline to input images. Open up a new file, name it `orient_images.py`, and insert the following code:

```
# import the necessary packages
from keras.applications import VGG16
from keras.applications import imagenet_utils
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import load_img
from imutils import paths
import numpy as np
import argparse
import pickle
import imutils
import h5py
import cv2
```

We start by importing our required Python packages. Notice how we'll once again need `VGG16` for feature extraction. We'll also load some helper utilities used to facilitate loading our input image from disk and preparing it for feature extraction by VGG16.

Next, let's parse our command line arguments:

```
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-d", "--db", required=True,
	help="path HDF5 database")
ap.add_argument("-i", "--dataset", required=True,
	help="path to the input images dataset")
ap.add_argument("-m", "--model", required=True,
	help="path to trained orientation model")
args = vars(ap.parse_args())
```

Our script requires three command line arguments, each of which are detailed below:

- `--db`: Here we supply the path to our HDF5 dataset of extracted features residing on disk. We *only* need this dataset so we can extract the label names (i.e., angles) from the HDF5 file, otherwise this switch would not be needed.
- `--dataset`: This is the path to our dataset of rotated images residing on disk.
- `--model`: Here we can provide the path to our trained Logistic Regression classifier used to predict the orientation of the images.

Let's go ahead and load the label names from our HDF5 dataset, followed by creating a sample of ten images from our input `--dataset` that we'll use to visualize our orientation correction:

```
# load the label names (i.e., angles) from the HDF5 dataset
db = h5py.File(args["db"])
labelNames = [int(angle) for angle in db["label_names"][:]]
db.close()

# grab the paths to the testing images and randomly sample them
print("[INFO] sampling images...")
```

```
31 imagePaths = list(paths.list_images(args["dataset"]))
32 imagePaths = np.random.choice(imagePaths, size=(10,), replace=False)
```

We'll also load the VGG16 network architecture and Logistic Regression classifier from disk as well:

```
34 # load the VGG16 network
35 print("[INFO] loading network...")
36 vgg = VGG16(weights="imagenet", include_top=False)
37
38 # load the orientation model
39 print("[INFO] loading model...")
40 model = pickle.loads(open(args["model"], "rb").read())
```

The core of the orientation correction pipeline can be found below:

```
42 # loop over the image paths
43 for imagePath in imagePaths:
44     # load the image via OpenCV so we can manipulate it after
45     # classification
46     orig = cv2.imread(imagePath)
47
48     # load the input image using the Keras helper utility while
49     # ensuring the image is resized to 224x224 pixels
50     image = load_img(imagePath, target_size=(224, 224))
51     image = img_to_array(image)
52
53     # preprocess the image by (1) expanding the dimensions and (2)
54     # subtracting the mean RGB pixel intensity from the ImageNet
55     # dataset
56     image = np.expand_dims(image, axis=0)
57     image = imagenet_utils.preprocess_input(image)
58
59     # pass the image through the network to obtain the feature vector
60     features = vgg.predict(image)
61     features = features.reshape((features.shape[0], 512 * 7 * 7))
```

On **Line 43** we loop over each of the individual sampled image paths. For each of these images, we load the original from disk via OpenCV (**Line 46**) so we can correct its orientation later in the script. **Lines 50 and 51** load the image from disk via the Keras helper utility, ensuring the channels of the image are properly ordered.

We then continue to pre-process the image for classification on **Lines 56 and 67**. **Line 60** performs a forward pass, passing the `image` through `vgg` and obtaining the output features. These feature are then flattened from a $512 \times 7 \times 7$ array into a flattened list of 9,975 floating point values. Flattening the output of the `POOL` layer is a requirement prior to passing the images through the Logistic Regression classifier, which we do below:

```
63     # now that we have the CNN features, pass these through our
64     # classifier to obtain the orientation predictions
65     angle = model.predict(features)
66     angle = labelNames[angle[0]]
```

Given the predicted `angle` from the input `features`, we can now correct the orientation of the image and display both the original and corrected image to our screen:

```
68      # now that we have the predicted orientation of the image we can
69      # correct for it
70      rotated = imutils.rotate_bound(orig, 360 - angle)
71
72      # display the original and corrected images
73      cv2.imshow("Original", orig)
74      cv2.imshow("Corrected", rotated)
75      cv2.waitKey(0)
```

To see our orientation correction script in action, execute the following command:

```
$ python orient_images.py --db indoor_cvpr/hdf5/orientation_features.hdf5 \
    --dataset indoor_cvpr/rotated_images --model models/orientation.cpickle
```

Figure 12.2: Correcting image orientation with features extracted from a pre-trained CNN.

The results of our algorithm can be seen in Figure 12.2. On the *left*, we have examples of two original, uncorrected images. Then on the *right*, we have our images *after* applying orientation correction. In each of these cases, we were able to *predict* the rotation angle of the original image and then correct it in the output image.

Algorithms and deep learning pipelines such as these can be used to process large datasets, perhaps of scanned images, where the orientation of the input images is unknown. Applying such a pipeline would *dramatically* reduce the amount of time taken for a human to manually correct the orientation prior to storing the image in an electronic database.

12.5 Summary

In this case study, we learned how to predict the orientation of images with no *a priori* knowledge. To accomplish this task, we generated a labeled dataset where images were randomly rotated by

$\{0, 90, 180, 360\}$ degrees. We then applied transfer learning via feature extraction and the VGG16 network architecture to extract features from the final `POOL` layer in the network. These features were fed into a Logistic Regression classifier, enabling us to correctly predict the orientation of an image with $\mathbf{93}\%$ accuracy. Combining *both* the VGG16 network and our trained Logistic Regression model enabled us to construct a pipeline that can automatically *detect* and *correct* image orientation.

Finally, it's once again worth mentioning that this result would have been impossible if our CNN was *truly* rotation invariant. CNNs are robust image classification tools and can correctly classify images under a variety of orientations; however, the individual filters inside the CNN are *not* rotation invariant.

Furthermore, CNNs are only able to classify objects under rotations on which they were trained. There is certainly a degree of generalization here as well; however, if these filters were *truly* rotation invariant the features we extracted from our input images would be near identical to each other. If the features were near identical, then the Logistic Regression classifier would not be able to discriminate between our image orientations. This is an important lesson to keep in mind when developing your own deep learning applications – if you expect input objects to exist under many orientations, make sure your training data reflects this requirement.

13. Case Study: Vehicle Identification

Have you ever seen the movie *Minority Report?* In the film, there is a scene where the protagonist (Tom Cruise) is walking through a crowded shopping mall. A combination of camera sensors mounted next to each LCD display, along with specialized computer vision algorithms, are able to (1) recognize who he is and (2) display targeted advertising based on his specific profile.

Back in 2002 when *Minority Report* was released, this type of robust, targeted advertising sounded like science fiction. But the truth is we aren't too far from it. Imagine you were driving a car down a highway filled with billboards – only instead of being posters plastered on a big wall, they were instead massive LCD screens. Then, based on what type of car you were driving, you would see a different ad. The interests of a person driving a brand new BMW 5 series might receive ads for golfing lessons, while the driver of a later model minivan might be drawn to advertising on how to save money for their kids' college. Whether or not these ads reflect the interest of the *individual* driver is questionable, but taken on aggregate, the following process is exactly how advertising works:

1. Analyze a demographic.
2. Identify their interests.
3. Display advertising that reflects these interests.

In this chapter, we'll learn how to build the computer vision component of an intelligent billboard system. Specifically, we'll learn how to fine-tune a pre-trained CNN with the mxnet library to recognize over 164 vehicle *makes* and *models* (using remarkably little training data) with over $\mathbf{96.54}\%$ accuracy.

13.1 The Stanford Cars Dataset

The Stanford cars dataset is a collection of 16,185 images of 196 cars (Figure 13.1), curated by Krause et al. in their 2013 publication, *3D Object Representation for Fine-Grained Classification* [37].

You can download an archive of the dataset here:
http://pyimg.co/9s9mx
After downloading, use the following command to uncompress the dataset:

Figure 13.1: The Stanford Cars Dataset consists of 16,185 images with 196 vehicle make and model classes.

```
$ tar -xvf car_ims.tar.gz
```

We'll discuss this dataset in more detail in Section 13.1.1. The goal of the original Krause et al. paper is to correctly predict the make, model, and year of a given vehicle. However, given that there are:

1. Extreme class imbalances in the dataset where some vehicle makes and models are *heavily* overrepresented (e.x., Audi and BMW *each* having over 1,000 data points while Tesla only has 77 examples).
2. Very little data, even for the large classes.

I decided to remove the year label and instead categorize images strictly based on their make and model. Even when doing categorization this way, there are instances where we have < 100 images per class, making it very challenging to fine-tune a deep CNN on this dataset. However, with the right fine-tuning approach, we'll still be able to obtain $> 95\%$ classification accuracy.

Once removing the year label, we were left with 164 total vehicle make and model classes to recognize. Our goal will be to fine-tune VGG16 to identify each of the 164 classes.

13.1.1 Building the Stanford Cars Dataset

In order to fine-tune VGG16 on the Stanford Cars dataset, we first need to compact the images into efficiently packed record files, just like in our chapters on training ImageNet. But before we get that far, let's take a look at the directory structure for our project:

```
--- car_classification
|    |--- config
|    |    |--- __init__.py
|    |    |--- car_config.py
|    |--- build_dataset.py
|    |--- fine_tune_cars.py
|    |--- test_cars.py
|    |--- vgg16
|    |    |--- vgg16-0000.params
|    |    |--- vgg16-symbol.json
|    |--- vis_classification.py
```

We'll once again have a `config` module where we'll store `car_config.py` – this file will

contain all necessary configurations to build the Stanford Cars Dataset and fine-tune VGG16 on it. Furthermore, there will be *substantially fewer* configuration variables required.

The `build_dataset.py`, as the name suggests, will input our image paths and class labels and then generate a `.lst` file for each of the training, validation, and testing splits, respectively. Once we have these `.lst` files, we can apply mxnet's `im2rec` binary to construct the record database.

The `fine_tune_cars.py` script will be responsible for fine-tuning VGG16 on our dataset. Once we have successfully fine-tuned VGG16, we'll want to evaluate the performance on the testing set – this is exactly what `test_cars.py` will accomplish.

Of course, evaluating a network on a pre-built dataset does not give you any insight in how to apply CNN trained with mxnet to a single, *individual* image. To investigate how to classify a single image futher (and demonstrate how you might build an intelligent ad-serving billboard), we'll create `vis_classification.py`. This script will load an input image from disk, pre-process it, pass it through our fine-tuned VGG16, and display the output predictions.

The Stanford Cars Configuration File

Before we can build our `.lst` and `.rec` files, we first need to create our `car_config.py` file. Go ahead and open up `car_config.py` now and we'll review the contents:

```
1  # import the necessary packages
2  from os import path
3
4  # define the base path to the cars dataset
5  BASE_PATH = "/raid/datasets/cars"
6
7  # based on the base path, derive the images path and meta file path
8  IMAGES_PATH = path.sep.join([BASE_PATH, "car_ims"])
9  LABELS_PATH = path.sep.join([BASE_PATH, "complete_dataset.csv"])
```

On **Line 5** I define the `BASE_PATH` to my Stanford Cars dataset. Inside the `BASE_PATH` you'll find my unarchived `cars_ims` directory, along with my `checkpoints`, `lists`, `output`, and `rec` directories used to store model checkpoints, dataset list files, model output, raw images, and `im2rec` encoded datasets:

```
$ ls
car_ims  car_ims.tgz  checkpoints  complete_dataset.csv  lists  output  rec
```

This configuration is how I *personally* prefer to define my directory structure for a given dataset, but you should configure your environment in the way you are most comfortable – just keep in mind that you will need to update the `BASE_PATH` and any other paths to match your system.

The `complete_dataset.csv` file is a custom CSV included with your download of *Deep Learning for Computer Vision with Python*. You can find the `complete_dataset.csv` file in the code directory associated with this chapter. Please either (1) copy `complete_dataset.csv` into your cars dataset as indicated in the directory structure above or (2) update the `LABELS_PATH` accordingly.

Line 8 uses the `BASE_PATH` to derive the `IMAGES_PATH`, which is where the input images to the Stanford Cars dataset live. As you can see, there are are 16,185 JPEG files, each of them named sequentially:

```
$ ls -l car_ims/*.jpg | wc -l
16185
$ ls -l car_ims/*.jpg | head -n 5
-rw-r--r-- 1 adrian adrian  466876 Feb 28  2015 car_ims/000001.jpg
-rw-r--r-- 1 adrian adrian   25139 Feb 28  2015 car_ims/000002.jpg
-rw-r--r-- 1 adrian adrian   19451 Feb 28  2015 car_ims/000003.jpg
-rw-r--r-- 1 adrian adrian   16089 Feb 28  2015 car_ims/000004.jpg
-rw-r--r-- 1 adrian adrian    1863 Feb 28  2015 car_ims/000005.jpg
```

Line 9 then defines the path to our `complete_dataset.csv` file. This file is actually *not* part of the dataset curated by Krause et al. Instead, original DevKit supplied by Krause et al. included MATLAB meta files that were hard to parse as *all three* vehicle make, model, *and* name were stored as a single string. To make the Stanford Cars dataset easier to work with, I created a `complete_dataset.csv` file which lists the image filename, vehicle make, model, model, type (sedan, coupe, etc.), and manufacture year all in a convenient comma separated file. Using this file, we'll be able to more easily build our dataset.

Next, we'll definite path to our output training, validation, and testing `.lst` files:

```
11 # define the path to the output training, validation, and testing
12 # lists
13 MX_OUTPUT = BASE_PATH
14 TRAIN_MX_LIST = path.sep.join([MX_OUTPUT, "lists/train.lst"])
15 VAL_MX_LIST = path.sep.join([MX_OUTPUT, "lists/val.lst"])
16 TEST_MX_LIST = path.sep.join([MX_OUTPUT, "lists/test.lst"])
```

As well as the paths to our `.rec` files for each of the data splits:

```
18 # define the path to the output training, validation, and testing
19 # image records
20 TRAIN_MX_REC = path.sep.join([MX_OUTPUT, "rec/train.rec"])
21 VAL_MX_REC = path.sep.join([MX_OUTPUT, "rec/val.rec"])
22 TEST_MX_REC = path.sep.join([MX_OUTPUT, "rec/test.rec"])
```

Notice how I am using the `path.sep` variable provided by Python to make these configurations as (reasonably) portable as possible. We'll need to encode the human readable vehicle make + model names as integers during training which will require a label encoder:

```
24 # define the path to the label encoder
25 LABEL_ENCODER_PATH = path.sep.join([BASE_PATH, "output/le.cpickle"])
```

This encoder will also enable us to perform the *inverse transform* and obtain the human readable names from the integer class labels.

In order to perform mean normalization we'll need to define the RGB means:

```
27 # define the RGB means from the ImageNet dataset
28 R_MEAN = 123.68
29 G_MEAN = 116.779
30 B_MEAN = 103.939
```

Keep in mind that these are the RGB means from the original Simonyan and Zisserman paper [17]. Since we are fine-tuning VGG, we must use the RGB means they derived from the *ImageNet* dataset – it would not make any sense for us to compute the RGB means for the cars dataset and then use them to fine-tune VGG. We need the *original* RGB means that VGG16 was trained on in order to successfully fine-tune the network to recognize vehicle makes and models.

Next, we need to define the total number of classes, along with the number of validation and testing images we'll be using:

```
32 # define the percentage of validation and testing images relative
33 # to the number of training images
34 NUM_CLASSES = 164
35 NUM_VAL_IMAGES = 0.15
36 NUM_TEST_IMAGES = 0.15
```

Our CNN must be able to recognize 164 different vehicle makes and models (**Line 34**). We'll then take 30% of the original dataset, and use 15% for validation and 15% for testing, leaving 70% for training.

Our final code block handles configuring the batch size and number of devices to train with:

```
38 # define the batch size
39 BATCH_SIZE = 32
40 NUM_DEVICES = 1
```

We'll use a mini-batch size of 32 images along with a single GPU to fine-tune VGG on the cars dataset. As we'll find out, fine-tuning VGG on the Stanford Cars dataset is *very fast* due to the small dataset – we'll be able to complete a *single epoch* in $\approx$ 10 minutes with just one GPU.

Creating the List Files

Now that our configuration file has been created, we can move on to the `build_dataset.py` script. Just like in our ImageNet experiments, this script will be responsible for building the training, validation, and testing `.lst` files. Go ahead and open up `build_dataset.py` and insert the following code:

```
1  # import the necessary packages
2  from config import car_config as config
3  from sklearn.preprocessing import LabelEncoder
4  from sklearn.model_selection import train_test_split
5  import progressbar
6  import pickle
7  import os
8
9  # read the contents of the labels file, then initialize the list of
10 # image paths and labels
11 print("[INFO] loading image paths and labels...")
12 rows = open(config.LABELS_PATH).read()
13 rows = rows.strip().split("\n")[1:]
14 trainPaths = []
15 trainLabels = []
```

Lines 2-7 import our required Python packages. We'll make sure to import our `car_config` module (aliased as `config`) so we can access our file paths and additional configurations. The

train_test_split function will be used to create our validation and testing splits, respectively. We then read the contents of the LABELS_PATH file (i.e., complete_dataset.csv) while initializing two lists: one for our training image paths and the other for our training labels.

Now that we have loaded the contents of our labels file, let's loop over each row individually:

```
17 # loop over the rows
18 for row in rows:
19     # unpack the row, then update the image paths and labels list
20     # (filename, make) = row.split(",")[:2]
21     (filename, make, model) = row.split(",")[:3]
22     filename = filename[filename.rfind("/") + 1:]
23     trainPaths.append(os.sep.join([config.IMAGES_PATH, filename]))
24     trainLabels.append("{}:{}".format(make, model))
```

For each row, we break the comma separated line into three values: the filename, vehicle make, and vehicle model. At this point the filename is represented as car_ims/000090.jpg, so we parse the path and remove the sub-directory, leaving the filename as 000090.jpg (**Line 22**). We update the trainPaths list with the path to our input training image by combining our IMAGES_PATH with the filename. The label is then constructed by concatenating the make and model, and then it's added to the trainLabels list.

Now that we have parsed the entire LABELS_PATH file, we need to compute the total number of validation and testing files we'll need based on the NUM_VAL_IMAGES and NUM_TEST_IMAGES percentages:

```
26 # now that we have the total number of images in the dataset that
27 # can be used for training, compute the number of images that
28 # should be used for validation and testing
29 numVal = int(len(trainPaths) * config.NUM_VAL_IMAGES)
30 numTest = int(len(trainPaths) * config.NUM_TEST_IMAGES)
```

We'll also take the time now to convert our strings labels to integers:

```
32 # our class labels are represented as strings so we need to encode
33 # them
34 print("[INFO] encoding labels...")
35 le = LabelEncoder().fit(trainLabels)
36 trainLabels = le.transform(trainLabels)
```

As mentioned earlier, we do not have a preset validation and testing set, so we need to create them from the training set. The following code block creates each of these splits:

```
38 # perform sampling from the training set to construct a a validation
39 # set
40 print("[INFO] constructing validation data...")
41 split = train_test_split(trainPaths, trainLabels, test_size=numVal,
42     stratify=trainLabels)
43 (trainPaths, valPaths, trainLabels, valLabels) = split
44
45 # perform stratified sampling from the training set to construct a
46 # a testing set
```

```
47 print("[INFO] constructing testing data...")
48 split = train_test_split(trainPaths, trainLabels, test_size=numTest,
49      stratify=trainLabels)
50 (trainPaths, testPaths, trainLabels, testLabels) = split
```

Next, let's build our `datasets` list where each entry is a 4-tuple containing the split type, paths to images, corresponding class labels, and path to output `.lst` file:

```
52 # construct a list pairing the training, validation, and testing
53 # image paths along with their corresponding labels and output list
54 # files
55 datasets = [
56      ("train", trainPaths, trainLabels, config.TRAIN_MX_LIST),
57      ("val", valPaths, valLabels, config.VAL_MX_LIST),
58      ("test", testPaths, testLabels, config.TEST_MX_LIST)]
```

To construct each `.lst` file, we need to loop over each of the entries in `datasets`:

```
60 # loop over the dataset tuples
61 for (dType, paths, labels, outputPath) in datasets:
62      # open the output file for writing
63      print("[INFO] building {}...".format(outputPath))
64      f = open(outputPath, "w")
65
66      # initialize the progress bar
67      widgets = ["Building List: ", progressbar.Percentage(), " ",
68           progressbar.Bar(), " ", progressbar.ETA()]
69      pbar = progressbar.ProgressBar(maxval=len(paths),
70           widgets=widgets).start()
```

Line 64 opens a file pointer to the current `outputPath`. We can then loop over each of the individual image paths and labels, writing them to the output `.lst` file as we do:

```
72      # loop over each of the individual images + labels
73      for (i, (path, label)) in enumerate(zip(paths, labels)):
74           # write the image index, label, and output path to file
75           row = "\t".join([str(i), str(label), path])
76           f.write("{}\n".format(row))
77           pbar.update(i)
78
79      # close the output file
80      pbar.finish()
81      f.close()
```

Our final code block handles saving the serialized label encoder to disk so we can reuse it later in this chapter:

```
83 # write the label encoder to file
84 print("[INFO] serializing label encoder...")
85 f = open(config.LABEL_ENCODER_PATH, "wb")
86 f.write(pickle.dumps(le))
87 f.close()
```

To build our training, validation, and testing `.lst` files, execute the following command:

```
$ python build_dataset.py
[INFO] loading image paths and labels...
[INFO] encoding labels...
[INFO] constructing validation data...
[INFO] constructing testing data...
[INFO] building /raid/datasets/cars/lists/train.lst...
Building List: 100% |###################################| Time: 0:00:00
[INFO] building /raid/datasets/cars/lists/val.lst...
Building List: 100% |###################################| Time: 0:00:00
[INFO] building /raid/datasets/cars/lists/test.lst...
Building List: 100% |###################################| Time: 0:00:00
[INFO] serializing label encoder...
```

After the command finishes running, you should verify that you indeed have the `train.lst`, `test.lst`, and `val.lst` files on your system:

```
$ wc -l lists/*.lst
  2427 lists/test.lst
 11331 lists/train.lst
  2427 lists/val.lst
 16185 total
```

Creating the Record Database

Given the `.lst` files of each of the splits, we now need to generate mxnet record files. Below you can see the command to generate the `train.rec` file which is the exact same file path as in `TRAIN_MX_REC` config:

```
$ ~/mxnet/bin/im2rec /raid/datasets/cars/lists/train.lst "" \
     /raid/datasets/cars/rec/train.rec resize=256 encoding='.jpg' \
     quality=100
...
[12:23:13] tools/im2rec.cc:298: Total: 11331 images processed, 124.967 sec elapsed
```

You can generate the `test.rec` dataset using this command:

```
$ ~/mxnet/bin/im2rec /raid/datasets/cars/lists/test.lst "" \
     /raid/datasets/cars/rec/test.rec resize=256 encoding='.jpg' \
     quality=100
...
[12:25:18] tools/im2rec.cc:298: Total: 2427 images processed, 33.5529 sec elapsed
```

And the `val.rec` file with this command:

```
$ ~/mxnet/bin/im2rec /raid/datasets/cars/lists/val.lst "" \
     /raid/datasets/cars/rec/val.rec resize=256 encoding='.jpg' \
          quality=100
...
[12:24:05] tools/im2rec.cc:298: Total: 2427 images processed, 23.9026 sec elapsed
```

As a sanity check, list the contents of the directory where you have stored the record databases:

```
$ ls -l rec
total 1188668
-rw-rw-r-- 1 adrian adrian 182749528 Aug 26 12:25 test.rec
-rw-rw-r-- 1 adrian adrian 851585528 Aug 26 12:23 train.rec
-rw-rw-r-- 1 adrian adrian 182850704 Aug 26 12:24 val.rec
```

As you can see from my output, my training database is approximately 850MB, while the validation and testing datasets are 181MB a piece, *substantially smaller* than the ImageNet dataset we worked with earlier.

13.2 Fine-tuning VGG on the Stanford Cars Dataset

We are now finally ready to fine-tune VGG16 on the Stanford Cars dataset. But before we can write the code for `fine_tune_cars.py`, we first need to download the *pre-trained* weights for VGG16. You can download the weights using this link:

http://data.dmlc.ml/models/imagenet/vgg/

Be sure to download both the `vgg16-0000.params` and `vgg16-symbol.json` files. Again, keep in mind that the weights for VGG16 are $\approx 550MB$, so be patient as the file downloads. The `.params` file are the actual *weights* while the `.json` file is the *architecture*. We'll learn how to load these two files and then fine-tune VGG16 in `fine_tune_cars.py`. Go ahead and open the file up now and we'll get to work:

```
1  # import the necessary packages
2  from config import car_config as config
3  import mxnet as mx
4  import argparse
5  import logging
6  import os
7
8  # construct the argument parse and parse the arguments
9  ap = argparse.ArgumentParser()
10 ap.add_argument("-v", "--vgg", required=True,
11      help="path to pre-trained VGGNet for fine-tuning")
12 ap.add_argument("-c", "--checkpoints", required=True,
13      help="path to output checkpoint directory")
14 ap.add_argument("-p", "--prefix", required=True,
15      help="name of model prefix")
16 ap.add_argument("-s", "--start-epoch", type=int, default=0,
17      help="epoch to restart training at")
18 args = vars(ap.parse_args())
```

Lines 2-6 import our Python packages – we'll import our `car_config` here so we can have access to our configurations. We then parse three required command line arguments, followed by a fourth optional one:

- `--vgg`: This is the path to the *pre-trained* VGG weights we will be fine-tuning.
- `--checkpoints`: During the fine-tuning process, we'll serialize the weights after every epoch. This switch controls where the serialized weight files will be stored.
- `--prefix`: Just like all other mxnet examples, we need to supply a name for our network.

- `--start-epoch`: If we choose to stop and restart training during fine-tuning, we can indicate which epoch we want to restart training from. This command line argument is *optional* provided you are fine-tuning from epoch zero.

Next, we'll set our logging file as well as determine our batch size based on the configuration file and the number of devices we'll be using for training:

```
20 # set the logging level and output file
21 logging.basicConfig(level=logging.DEBUG,
22     filename="training_{}.log".format(args["start_epoch"]),
23     filemode="w")
24
25 # determine the batch
26 batchSize = config.BATCH_SIZE * config.NUM_DEVICES
```

In order to access our training data, we'll need to instantiate the `ImageRecordIter`:

```
28 # construct the training image iterator
29 trainIter = mx.io.ImageRecordIter(
30     path_imgrec=config.TRAIN_MX_REC,
31     data_shape=(3, 224, 224),
32     batch_size=batchSize,
33     rand_crop=True,
34     rand_mirror=True,
35     rotate=15,
36     max_shear_ratio=0.1,
37     mean_r=config.R_MEAN,
38     mean_g=config.G_MEAN,
39     mean_b=config.B_MEAN,
40     preprocess_threads=config.NUM_DEVICES * 2)
```

The VGG16 network architecture assumes input images that are 224×224 pixels with 3 channels; however, recall from Section 13.1.1 above, that we created `.rec` files with images that are 256 pixels along their shortest dimension . Why did we do create these files? The answer lies with the `rand_crop` attribute (**Line 33**). This attribute indicates that we wish to *randomly crop* 224×224 regions from the 256×256 input image. Doing so can help us improve classification accuracy. We'll also supply the RGB means from the *original* VGG16 trained by Simonyan and Zisserman since we are *fine-tuning* rather than training from scratch.

Let's also construct the validation image iterator:

```
42 # construct the validation image iterator
43 valIter = mx.io.ImageRecordIter(
44     path_imgrec=config.VAL_MX_REC,
45     data_shape=(3, 224, 224),
46     batch_size=batchSize,
47     mean_r=config.R_MEAN,
48     mean_g=config.G_MEAN,
49     mean_b=config.B_MEAN)
```

In order to fine-tune VGG16, we'll be using the SGD optimizer:

```
51 # initialize the optimizer and the training contexts
52 opt = mx.optimizer.SGD(learning_rate=1e-4, momentum=0.9, wd=0.0005,
53      rescale_grad=1.0 / batchSize)
54 ctx = [mx.gpu(3)]
```

We'll be using a small(er) learning rate of $1e-4$ initially – future experiments will help us determine what the optimal initial learning rate is. We'll also train with a momentum of 0.9 and L2 weight regularization term of 0.0005. A single GPU will be used to fine-tune VGG on the Stanford Cars dataset as epochs will only take ≈ 11 seconds.

Next, we'll build the path to our output `checkpointsPath`, as well as initialize the argument parameters, auxiliary parameters, and whether or not "missing" parameters are allowed in the network:

```
56 # construct the checkpoints path, initialize the model argument and
57 # auxiliary parameters, and whether uninitialized parameters should
58 # be allowed
59 checkpointsPath = os.path.sep.join([args["checkpoints"],
60      args["prefix"]])
61 argParams = None
62 auxParams = None
63 allowMissing = False
```

In this context, "missing parameters" are parameters that have not yet been initialized in the network. Normally we would *not* allow uninitialized parameters; however, recall that fine-tuning requires us to slice off the head of the network and replace it with a new, uninitialized fully-connected head. Therefore, if we are training from epoch zero, we *will* allow missing parameters.

Speaking of training from epoch zero, let's see how this process is done:

```
65 # if there is no specific model starting epoch supplied, then we
66 # need to build the network architecture
67 if args["start_epoch"] <= 0:
68      # load the pre-trained VGG16 model
69      print("[INFO] loading pre-trained model...")
70      (symbol, argParams, auxParams) = mx.model.load_checkpoint(
71           args["vgg"], 0)
72      allowMissing = True
```

Line 67 makes a check to see if we are starting the initial fine-tuning process. Provided that we are, **Lines 70 and 71** load the pre-trained `--vgg` weights from file. We also update `allowMissing` to be `True` as we're about to replace the head of the network.

Replacing the head of the network is not as straightforward as Keras (as it involves some investigative work in the layer names), but it's still a relatively straightforward process:

```
74      # grab the layers from the pre-trained model, then find the
75      # dropout layer *prior* to the final FC layer (i.e., the layer
76      # that contains the number of class labels)
77      # HINT: you can find layer names like this:
78      # for layer in layers:
79      #      print(layer.name)
```

```
80      # then, append the string '_output' to the layer name
81      layers = symbol.get_internals()
82      net = layers["drop7_output"]
```

Line 81 grabs all `layers` inside the VGG16 network we just loaded from disk. We then need to determine where the final *output* layer is and then slice off the rest of the `FC` head. This process requires a bit of investigative work as we need to know the *name* of the final output layer we are interested in (in this case, the final dropout layer before we apply the 1,000 node `FC` layer for ImageNet labels). To find this dropout layer, I recommend loading VGG16 in a separate Python shell, grabbing the layers, and printing out the names:

```
$ python
>>> import mxnet as mx
>>> (symbol, argParams, auxParams) = mx.model.load_checkpoint("vgg16", 0)
>>> layers = symbol.get_internals()
>>> for layer in layers:
...     print(layer.name)
...
data
conv1_1_weight
conv1_1_bias
conv1_1
relu1_1
conv1_2_weight
conv1_2_bias
conv1_2
relu1_2
pool1
...
fc7_weight
fc7_bias
fc7
relu7
drop7
fc8_weight
fc8_bias
fc8
prob_label
prob
```

Here we can see that `drop7` is the final dropout layer before the 1,000 node `FC` layer. To obtain the output of this class, simply append `output` to the end of the `drop7` string: `drop7_output`. Constructing this string serves as the key to our `layers` dictionary on **Line 82**, allowing us to keep all layers *up until* the final `FC` layer.

Our next code block attaches a new `FC` head with `NUM_CLASSES` (164) to the body of VGG16, followed by a softmax classifier:

```
84      # construct a new FC layer using the desired number of output
85      # class labels, followed by a softmax output
86      net = mx.sym.FullyConnected(data=net,
87           num_hidden=config.NUM_CLASSES, name="fc8")
88      net = mx.sym.SoftmaxOutput(data=net, name="softmax")
```

```
89
90        # construct a new set of network arguments, removing any previous
91        # arguments pertaining to FC8 (this will allow us to train the
92        # final layer)
93        argParams = dict({k:argParams[k] for k in argParams
94            if "fc8" not in k})
```

Lines 93 and 94 delete any parameter entries for `fc8`, the `FC` layer we just surgically removed from the network. The problem is that `argParams` does not contain any information regarding our *new* `FC` head, which is exactly why we set `allowMissing` to `True` earlier in the code.

In the case that we are restarting our fine-tuning from a specific epoch, we simply need to load the respective weights:

```
96  # otherwise, a specific checkpoint was supplied
97  else:
98        # load the checkpoint from disk
99        print("[INFO] loading epoch {}...".format(args["start_epoch"]))
100       (net, argParams, auxParams) = mx.model.load_checkpoint(
101           checkpointsPath, args["start_epoch"])
```

Let's also initialize our standard set of callbacks and metrics:

```
103 # initialize the callbacks and evaluation metrics
104 batchEndCBs = [mx.callback.Speedometer(batchSize, 50)]
105 epochEndCBs = [mx.callback.do_checkpoint(checkpointsPath)]
106 metrics = [mx.metric.Accuracy(), mx.metric.TopKAccuracy(top_k=5),
107       mx.metric.CrossEntropy()]
```

Finally, we can fine-tune our network:

```
109 # construct the model and train it
110 print("[INFO] training network...")
111 model = mx.mod.Module(symbol=net, context=ctx)
112 model.fit(
113       trainIter,
114       eval_data=valIter,
115       num_epoch=65,
116       begin_epoch=args["start_epoch"],
117       initializer=mx.initializer.Xavier(),
118       arg_params=argParams,
119       aux_params=auxParams,
120       optimizer=opt,
121       allow_missing=allowMissing,
122       eval_metric=metrics,
123       batch_end_callback=batchEndCBs,
124       epoch_end_callback=epochEndCBs)
```

The call the `fit` method in our fine-tuning example is a bit more verbose than the previous ImageNet examples because most of our previous parameters (e.x., `arg_params`, `aux_params`, etc.) could be passed in via the `FeedForward` class. However, since we are now relying on either (1)

performing network surgery or (2) loading a specific epoch, we need to move all these parameters into the `fit` method.

Also, notice how we supply the `allow_missing` parameter to mxnet, enabling the library to understand that we are attempting to perform fine-tuning. We'll set our maximum `num_epoch` to 65 – this number may increase or decrease depending on how the training process is going.

13.2.1 VGG Fine-tuning: Experiment #1

In my first experiment, fine-tuning the VGG16 architecture on the Stanford Cars dataset, I decided to use the SGD optimizer with an initial learning rate of $1e-4$. This is technically a large learning rate for a fine-tuning task, but I decided to give it a try to obtain a baseline accuracy. A momentum term of 0.9 and L2 weight decay of 0.0005 were also supplied. I then started training using the following command:

```
$ python fine_tune_cars.py --vgg vgg16/vgg16 --checkpoints checkpoints \
    --prefix vggnet
```

During the first 10 epochs, we can see that our fine-tuning looks quite good, with nearly 70% rank-1 accuracy and over 90% rank-5 accuracy being obtained (Figure 13.2, *left*). However, past epochs 15-30, we start to see VGG overfitting to the training data.

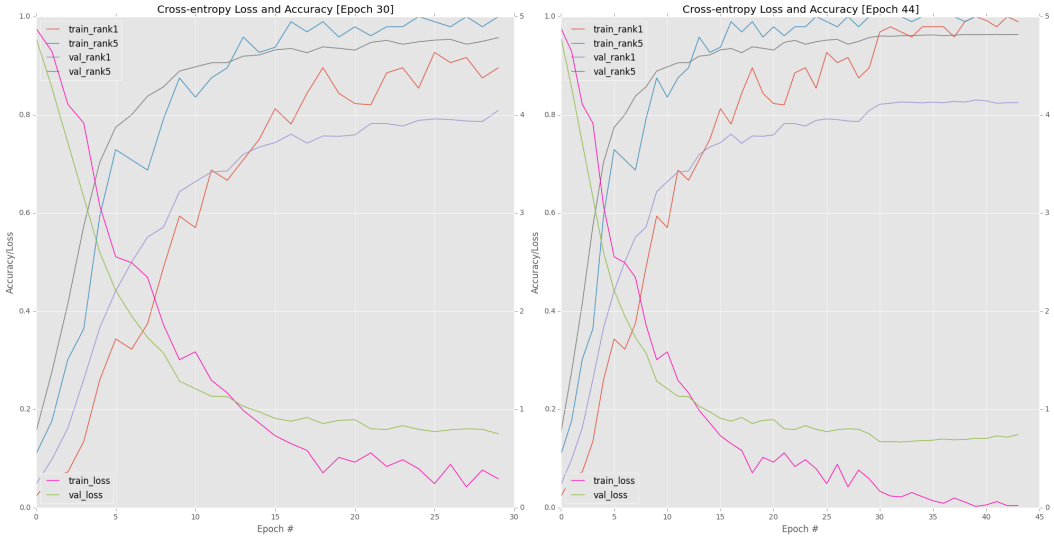


Figure 13.2: **Left:** Fine-tuning VGG16 on the cars dataset. The first ten epochs look good, but past epoch 15 we start to see overfitting. **Right:** Lowering α to $1e-5$ saturates learning.

I decided to stop training at this point and lower the learning rate from $1e-4$ to $1e-5$ and then restart training command:

```
$ python fine_tune_cars.py --vgg vgg16/vgg16 --checkpoints checkpoints \
    --prefix vggnet --start-epoch 30
```

I then let training resume for 15 more epochs until epoch 45 (Figure 13.2, *right*). As we can see, both training loss and accuracy are nearly fully saturated – the rank-1 accuracy for the training data is actually higher than the rank-5 accuracy for the validation set. Furthermore, examining

our cross-entropy loss plot, we can see that validation loss is starting to increase past epoch 35, a sure-fire sign of overfitting.

All that said, this initial experiment obtained 82.48% rank-1 and 96.38% rank-5 accuracy. The problem is that our network is far too overfit and we need to explore other avenues.

13.2.2 VGG Fine-tuning: Experiment #2

Due to the overfitting caused by the base learning rate of $1e-4$, I decided to lower my initial learning rate to $1e-5$. SGD was once again used as my optimizer with the same momentum and weight decay parameters as the first experiment. Training was started using the following command:

```
$ python fine_tune_cars.py --vgg vgg16/vgg16 --checkpoints checkpoints \
    --prefix vggnet
```

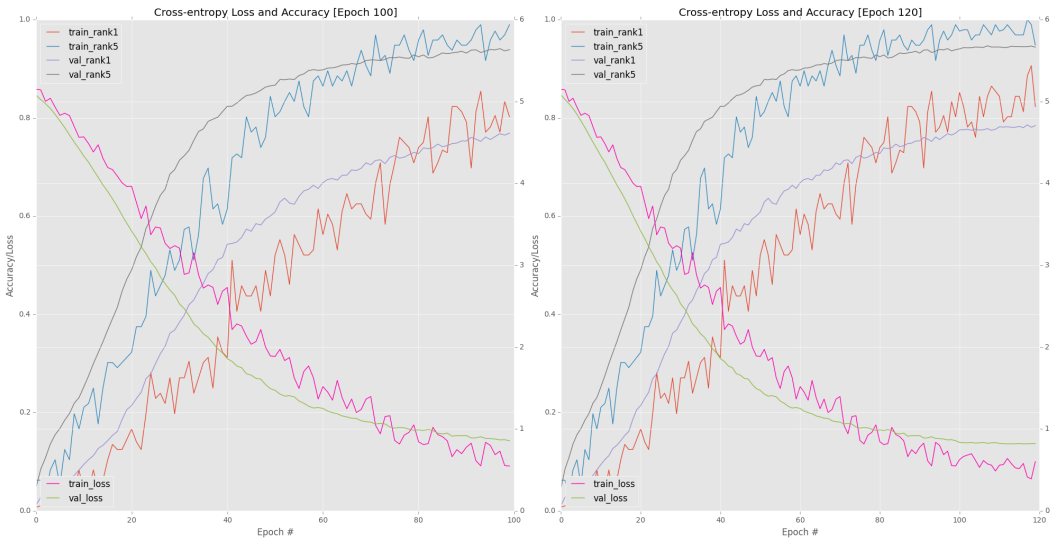


Figure 13.3: **Left:** Learning is much more stable using a $1e-5$ learning rate. **Right:** However, accuracy is much lower than our previous experiment, even when reducing α to $1e-6$. The initial learning rate is too low in this case.

Examining the plots of the first 80 epochs, we can see that learning is much more stable (Figure 13.3, *left*). The effects of overfitting have been reduced. The problem is that it's taking a long time for the network to train – it's also questionable if our validation accuracy will be as high as the previous experiment. At epoch 100 I stopped training, lowered the learning rate from $1e-5$ to $1e-6$, then resumed training:

```
$ python fine_tune_cars.py --vgg vgg16/vgg16 --checkpoints checkpoints \
    --prefix vggnet --start-epoch 100
```

I allowed the network to continue training for another 20 epochs to epoch 120 before I stopped the experiment (Figure 13.3, *right*). The problem here is that accuracy has dropped significantly – down to 78.41% rank-1 and 94.47% rank-5. A base learning rate of $1e-5$ was *too low* and therefore hurt our classification accuracy.

13.2.3 VGG Fine-tuning: Experiment #3

In my final VGG16 fine-tuning experiment, I decided to split the difference between the $1e-4$ and $1e-5$ learning rate and start training with a $5e-4$ learning rate. The same SGD optimizer, momentum, and regularization terms were kept as the first two experiments. I once again started the fine-tuning process using the following command:

```
$ python fine_tune_cars.py --vgg vgg16/vgg16 --checkpoints checkpoints \
    --prefix vggnet
```

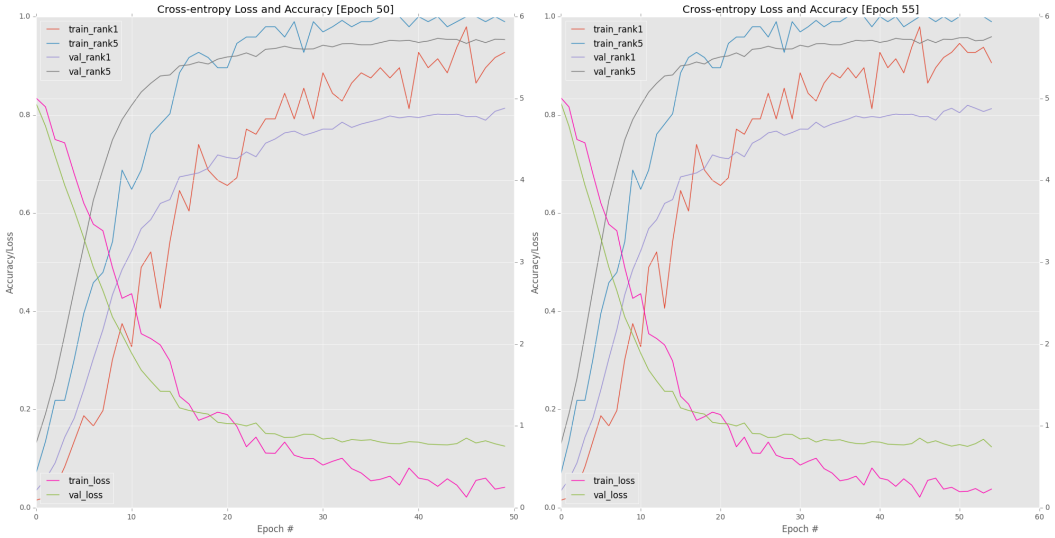


Figure 13.4: **Left:** The first 50 epochs using a $5e-4$ learning rate. **Right:** Reducing the learning rate to $5e-5$ and training for another 5 epochs. We obtain slightly lower accuracy than Experiment #1; however, we are *much* less overfit.

The first 50 epochs are plotted in Figure 13.4 *(left)*. Here we can see the training loss/accuracy keeping pace with the validation, and eventually overtaking the validation loss/accuracy. However, what's important to note here is that the training loss/accuracy is not *saturating* as it did in our first experiments.

I allowed the network to continue training until epoch 50 where I stopped training, reduced the learning rate from $5e-4$ to $5e-5$ and trained for another five epochs:

```
$ python fine_tune_cars.py --vgg vgg16/vgg16 --checkpoints checkpoints \
    --prefix vggnet --start-epoch 50
```

The final loss/accuracy plot can be seen in Figure 13.4 *(right)*. While there is certainly a gap between the training and validation loss starting to form, the validation loss is not increasing. Looking at the output of the final epoch, our network was reaching 81.25% rank-1 and 95.88% rank-5 accuracy on the validation set. These accuracies are certainly better than our second experiment. I would also argue that these accuracies are more *desirable* than the first experiment as we did not overfit our network and risk the ability of the model to generalize. At this point, I felt comfortable stopping the experiment altogether and moving on to the evaluation stage.

Keep in mind that when fine-tuning (large) networks such as VGG16 on small datasets (such as Stanford Cars), that overfitting is an inevitability. Even when applying data augmentation there are simply too many parameters in the network and too few training examples. Thus, it's *extremely important* that you get the initial learning rate correct – this is even more critical than when training a network from scratch. Take your time when fine-tuning networks and explore a variety of initial learning rates. This process will give you the best chance at obtaining the highest accuracy during fine-tuning.

13.3 Evaluating our Vehicle Classifier

Now that we have successfully fine-tuned VGG16 on the Stanford Cars Dataset, let's move on to evaluating the performance of our network on the testing set. To accomplish this process, open up the `test_cars.py` file and insert the following code:

```
1  # import the necessary packages
2  from config import car_config as config
3  from pyimagesearch.utils.ranked import rank5_accuracy
4  import mxnet as mx
5  import argparse
6  import pickle
7  import os
```

Lines 1-7 import our required Python packages. We'll need our `car_config` (aliased as `config`) so we can access our car dataset specific configuration and variables (**Line 2**). We'll import the `rank5_accuracy` function on **Line 3** so we can compute the rank-1 and rank-5 accuracy on the testing set, respectively. The `mxnet` library is then imported on **Line 4** so we can have access to the mxnet Python bindings.

Next, let's parse our command line arguments:

```
9  # construct the argument parse and parse the arguments
10 ap = argparse.ArgumentParser()
11 ap.add_argument("-c", "--checkpoints", required=True,
12      help="path to output checkpoint directory")
13 ap.add_argument("-p", "--prefix", required=True,
14      help="name of model prefix")
15 ap.add_argument("-e", "--epoch", type=int, required=True,
16      help="epoch # to load")
17 args = vars(ap.parse_args())
```

Similar to our experiments evaluating ImageNet, our command line arguments are near identical. To start, we need `--checkpoints`, which is the path to the directory where VGG16 weights were serialized during the fine-tuning process. The `--prefix` controls the name of the network, which in this case will be `vggnet`. Finally, `--epoch` is an integer which controls the weight epoch of VGG16 we are going to load for evaluation.

From there, we can load our label encoder as well as initialize the `ImageRecordIter` for the testing set:

```
19 # load the label encoder
20 le = pickle.loads(open(config.LABEL_ENCODER_PATH, "rb").read())
21
```

```
22 # construct the validation image iterator
23 testIter = mx.io.ImageRecordIter(
24      path_imgrec=config.TEST_MX_REC,
25      data_shape=(3, 224, 224),
26      batch_size=config.BATCH_SIZE,
27      mean_r=config.R_MEAN,
28      mean_g=config.G_MEAN,
29      mean_b=config.B_MEAN)
```

The next step is to then load our pre-trained model from disk:

```
31 # load our pre-trained model
32 print("[INFO] loading pre-trained model...")
33 checkpointsPath = os.path.sep.join([args["checkpoints"],
34      args["prefix"]])
35 (symbol, argParams, auxParams) = mx.model.load_checkpoint(
36      checkpointsPath, args["epoch"])
37
38 # construct the model
39 model = mx.mod.Module(symbol=symbol, context=[mx.gpu(0)])
40 model.bind(data_shapes=testIter.provide_data,
41      label_shapes=testIter.provide_label)
42 model.set_params(argParams, auxParams)
```

Lines 33 and 34 construct the base path to the serialized weights using both the `--checkpoints` and `--prefix` switch. To control exactly *which* epoch is loaded, we pass in the `checkpointsPath` and `--epoch` to the `load_checkpoint` function (**Lines 35 and 36**).

Line 39 creates the mxnet `Module` class. This class accepts two parameters, the model `symbol` loaded from disk along with the `context`, which is the list of devices we will use to evaluate our network on the `testIter` dataset. Here I indicate that only a single GPU is need via `mx.gpu(0)`; however, you should update this parameter to be the number of devices on your system.

A call to the `bind` method of `model` on **Line 40 and 41** is required in order for the `model` to understand that the `testIter` is responsible for providing the shape (i.e., dimensions) of both the data and labels. In our case, all input images will be 224×224 with three channels. Finally, we set the argument parameters and auxiliary parameters of the serialized network on **Line 42**.

At this point, we can start evaluating the network on the testing set:

```
44 # initialize the list of predictions and targets
45 print("[INFO] evaluating model...")
46 predictions = []
47 targets = []
48
49 # loop over the predictions in batches
50 for (preds, _, batch) in model.iter_predict(testIter):
51      # convert the batch of predictions and labels to NumPy
52      # arrays
53      preds = preds[0].asnumpy()
54      labels = batch.label[0].asnumpy().astype("int")
55
56      # update the predictions and targets lists, respectively
57      predictions.extend(preds)
58      targets.extend(labels)
```

```
59
60  # apply array slicing to the targets since mxnet will return the
61  # next full batch size rather than the *actual* number of labels
62  targets = targets[:len(predictions)]
```

Lines 46 and 47 initialize our list of `predictions` and ground-truth `targets`. On **Line 50** we use the `iter_predict` function to allow us to loop over the `testIter` in *batches*, yielding us both the predictions (`preds`) and the ground-truth labels (`batch`).

On **Line 53** we grab the predictions from the network and convert it to a NumPy array. This array has the shape of `(N, 164)` where N is the number of data points in the batch and 164 is the total number of class labels. **Line 54** extracts the ground-truth class label from the `testIter`, again converting it to a NumPy array. Given both the `preds` and ground-truth `labels`, we are now able to update our `predictions` and `targets` lists, respectively.

Line 62 ensures that both the `targets` and `predictions` lists are the same length. This line is a requirement as the `iter_predict` function will only return `batch` in sizes of powers of two for efficiency reasons (or it could also be a small bug in the function). Thus, it's nearly always the case that the `targets` list is longer than the `predictions` list. We can easily fix the discpreancy by applying array slicing.

The final step is to take our `predictions` and `targets` from the testing set and compute our rank-1 and rank-5 accuracies:

```
64  # compute the rank-1 and rank-5 accuracies
65  (rank1, rank5) = rank5_accuracy(predictions, targets)
66  print("[INFO] rank-1: {:.2f}%".format(rank1 * 100))
67  print("[INFO] rank-5: {:.2f}%".format(rank5 * 100))
```

To evaluate our fine-tuned VGG16 network on the vehicle make and model dataset, just execute the following command:

```
$ python test_cars.py --checkpoints checkpoints --prefix vggnet \
    --epoch 55
[INFO] loading pre-trained model...
[INFO] evaluating model...
[INFO] rank-1: 84.22%
[INFO] rank-5: 96.54%
```

As the results demonstrate, we are able to obtain **84.22%** rank-1 and **96.54%** rank-5 accuracy on the testing set. This level of accuracy is especially impressive given that we (1) have a *very* limited amount of training data for each vehicle make and model combination and (2) need to predict a fairly large number of classes (164) given this small amount of training data.

Further accuracy can be obtained by gathering *more* training data for each vehicle make and model. Given this extra data, we could either train a custom CNN from scratch or continue to apply fine-tuning. Since we are already performing very well with fine-tuning, I would focus efforts on data gathering and performing more fine-tuning experiments to boost accuracy.

13.4 Visualizing Vehicle Classification Results

In our previous experiment, we learned how to use the `iter_predict` function to loop over the data points in an `ImageRecordIter` and make predictions. But what if we wanted to loop over

raw images and make predictions on those? What are we to do then? Luckily for us, the process of preparing an image for classification with mxnet is not unlike the methods we used for Keras.

In this section, we'll learn how to load *individual* images from disk, pre-process them, and pass them through mxnet to obtain our top-5 predictions for a given input vehicle. To see how this is done, open up the `vis_classification.py` file and insert the following code:

```
1  # due to mxnet seg-fault issue, need to place OpenCV import at the
2  # top of the file
3  import cv2
4
5  # import the necessary packages
6  from config import car_config as config
7  from pyimagesearch.preprocessing import ImageToArrayPreprocessor
8  from pyimagesearch.preprocessing import AspectAwarePreprocessor
9  from pyimagesearch.preprocessing import MeanPreprocessor
10 import numpy as np
11 import mxnet as mx
12 import argparse
13 import pickle
14 import imutils
15 import os
```

On **Line 3** we import our `cv2` bindings to the OpenCV library. On two out of the three machines I installed both OpenCV and mxnet on, I received a segmentation fault error whenever I tried to import mxnet *before* OpenCV. Because of this error, I recommend placing your `cv2` import at the top of your file. That said, it may not be the case that your system seg-faults, so feel free to test this out on your own machine.

From there, **Lines 6-15** import the rest of our Python packages. Take note of **Lines 7-9** where we import our image pre-processors, initially designed for Keras, but they can be easily reused when working with mxnet as well.

Next, let's parse our command line arguments:

```
17 # construct the argument parse and parse the arguments
18 ap = argparse.ArgumentParser()
19 ap.add_argument("-c", "--checkpoints", required=True,
20     help="path to the checkpoint directory")
21 ap.add_argument("-p", "--prefix", required=True,
22     help="name of model prefix")
23 ap.add_argument("-e", "--epoch", type=int, required=True,
24     help="epoch # to load")
25 ap.add_argument("-s", "--sample-size", type=int, default=10,
26     help="epoch # to load")
27 args = vars(ap.parse_args())
```

We'll require three command line arguments to run `vis_classification.py`:

- `--checkpoints`: The path to our VGG16 checkpoints directory when we fine-tuned.
- `--prefix`: The name of our network that we fine-tuned.
- `--epoch`: The epoch number that we will be loading from disk.

Finally, an optional switch, `--sample-size`, can be supplied to indicate the number of images we wish to sample from our dataset for classification.

Speaking of sampling our images, let's go ahead and do that now:

```
29 # load the label encoder, followed by the testing dataset file,
30 # then sample the testing set
31 le = pickle.loads(open(config.LABEL_ENCODER_PATH, "rb").read())
32 rows = open(config.TEST_MX_LIST).read().strip().split("\n")
33 rows = np.random.choice(rows, size=args["sample_size"])
```

Line 31 loads our serialized `LabelEncoder` so we can convert the integer class labels produced by mxnet and convert them to human readable labels (i.e., the make and model names). **Lines 32 and 33** load the contents of our `test.lst` file and sample `--sample-size` rows from them. Recall that the `test.lst` file contains the paths to our testing images. Therefore, to visualize our predictions, we simply need to sample rows from this file.

We can then load our serialized VGG16 network from disk:

```
35 # load our pre-trained model
36 print("[INFO] loading pre-trained model...")
37 checkpointsPath = os.path.sep.join([args["checkpoints"],
38      args["prefix"]])
39 model = mx.model.FeedForward.load(checkpointsPath,
40      args["epoch"])
```

As well as compile the model:

```
42 # compile the model
43 model = mx.model.FeedForward(
44      ctx=[mx.gpu(0)],
45      symbol=model.symbol,
46      arg_params=model.arg_params,
47      aux_params=model.aux_params)
```

From there, let's initialize our image pre-processors:

```
49 # initialize the image pre-processors
50 sp = AspectAwarePreprocessor(width=224, height=224)
51 mp = MeanPreprocessor(config.R_MEAN, config.G_MEAN, config.B_MEAN)
52 iap = ImageToArrayPreprocessor(dataFormat="channels_first")
```

Line 50 instantiates our `AspectAwarePreprocessor` which will resize images to 224×224 pixels. The `MeanPreprocessor` will perform mean subtraction using the RGB averages from the Simonyan and Zisserman paper [17] – the means are *not* computed from our training set as we are doing fine-tuning and must use the averages computed over the ImageNet dataset.

Finally, we initialize our `ImageToArrayPreprocessor` on **Line 52**. Originally, this class was used to convert raw images to Keras-compatible arrays based on whether we were using "channels last" or "channels first" in our `keras.json` configuration file. However, since mxnet *always* represents images in channels first ordering, we need to supply the `dataFormat="channels_first"` parameter to the class to ensure our channels are ordered properly.

It's time to loop over our sample images, classify them, and display the results to our screen:

```
54 # loop over the testing images
55 for row in rows:
56     # grab the target class label and the image path from the row
57     (target, imagePath) = row.split("\t")[1:]
58     target = int(target)
59
60     # load the image from disk and pre-process it by resizing the
61     # image and applying the pre-processors
62     image = cv2.imread(imagePath)
63     orig = image.copy()
64     orig = imutils.resize(orig, width=min(500, orig.shape[1]))
65     image = iap.preprocess(mp.preprocess(sp.preprocess(image)))
66     image = np.expand_dims(image, axis=0)
```

Line 57 and 58 extract the ground-truth `target` and `imagePath` from the input `row`, followed by converting the `target` to an integer. We then:

1. Load our input `image` (**Line 62**).
2. Clone it so we can draw the output class label visualization on it (**Line 63**).
3. Start the pre-processing stage by resizing the image to have a maximum width of 500 pixels (**Line 64**).
4. Apply all three of our image pre-processors (**Line 65**).
5. Expand the dimensions of the array so the image can be passed through the network (**Line 66**).

Classifying an image using a pre-trained mxnet network is as simple as calling the `predict` method of `model`:

```
68     # classify the image and grab the indexes of the top-5 predictions
69     preds = model.predict(image)[0]
70     idxs = np.argsort(preds)[::-1][:5]
71
72     # show the true class label
73     print("[INFO] actual={}".format(le.inverse_transform(target)))
74
75     # format and display the top predicted class label
76     label = le.inverse_transform(idxs[0])
77     label = label.replace(":", " ")
78     label = "{}: {:.2f}%".format(label, preds[idxs[0]] * 100)
79     cv2.putText(orig, label, (10, 30), cv2.FONT_HERSHEY_SIMPLEX,
80         0.6, (0, 255, 0), 2)
```

Line 69 returns our predictions for each of the 164 class labels. We then sort the indexes of these labels according to their probability (from largest to smallest), keeping the top-5 predictions (**Line 70**). **Line 73** displays the human readable class name of the ground-truth label, while **Lines 76-80** draw the #1 *predicted* label to our image, including the probability of the prediction.

Our last code block handles printing the top-5 predictions to our terminal and displaying our output image:

```
82     # loop over the predictions and display them
83     for (i, prob) in zip(idxs, preds):
84         print("\t[INFO] predicted={}, probability={:.2f}%".format(
85             le.inverse_transform(i), preds[i] * 100))
```

```
86
87      # show the image
88      cv2.imshow("Image", orig)
89      cv2.waitKey(0)
```

To see `vis_classification.py` in action, simply open up a terminal and execute the following command:

```
$ python vis_classification.py --checkpoints checkpoints --prefix vggnet \
    --epoch 55
```

Figure 13.5: Our fine-tuned VGG16 network can correctly recognize the *make* and *model* of a vehicle with over 84% rank-1 and 95% rank-5 accuracy.

In Figure 13.5 you can see samples of correctly classified vehicle make and models. Again, it's quite remarkable that we are able to obtain such high classification accuracy using *such little* training data via fine-tuning. This is yet another example of the power and utility of deep learning applied to image classification.

13.5 Summary

In this chapter, we learned how to fine-tune VGG16 architecture (pre-trained on ImageNet) to correctly recognize 164 vehicle make and model classes with over 84.22% rank-1 and 96.54% rank-5 testing accuracy. To accomplish this task, we sliced off the final fully-connected layer in VGG16 (the `FC` layer that outputs the total number of classes, which is 1,000 in the case of ImageNet) and replaced it with our own fully-connected layer of 164 classes, followed by a softmax classifier.

A series of experiments were performed to determine the optimal learning rate when fine-tuning VGG16 using the SGD optimizer. After fine-tuning and evaluating our network, we then wrote a simple Python script named `vis_classification.py` to help us visualize the vehicle make and model classifications for a given input image. This same script can be modified to make intelligent, highly targeted billboard advertisements of your own, strictly based on the type of car a given person is driving.

14. Case Study: Age and Gender Prediction

In this final Case Study in *Deep Learning for Computer Vision with Python*, we'll learn how to build a computer vision system capable of recognizing the *age* and *gender* of a person in a photograph. To accomplish this task, we'll be using the Adience dataset, curated by Levi and Hassner and used in their 2015 publication, *Age and Gender Classification using Convolutional Neural Networks* [38].

This case study will be more complex than the previous ones, requiring us to train *two* models, one for age recognition and another for gender identification. Furthermore, we'll also have to rely on more advanced computer vision algorithms such as facial landmarks (http://pyimg.co/xkgwd) and face alignment (http://pyimg.co/tnbzf) to help us pre-process our images prior to classification.

We'll start off this chapter with a discussion on the Adience dataset and then review our project structure. From there, we'll create `MxAgeGenderNet`, an implementation of the network proposed by Levi et al. Given our implementation, we'll train two separate instantiations of `MxAgeGenderNet`, one to recognize ages and the other to recognize genders. Once we have these two networks, we'll apply them to images *outside* the Adience dataset and evaluate performance.

Due to the length of this chapter, we'll review the most important aspects of the case study, including reviewing the code used to train and evaluate the network. Further detail, especially regarding the utility functions and helper class we'll define later in the chapter can be found in the supplementary material accompanying this book.

14.1 The Ethics of Gender Identification in Machine Learning

Before we get too far in this chapter I want to bring up the subject of *ethics* and *gender identification*.

While utilizing computer vision and deep learning to recognize the age and gender of a person in a photo is an interesting classification challenge, it is one wrought with *moral implications*. Just because someone visually *looks*, *dresses*, or *appears* a certain way ***does not*** imply that they identify with that (or any) gender.

Before even considering building a piece of software that attempts to recognize the gender of a person, take the time to educate yourself on gender issues and gender equality.

Applications such as gender recognition are best used for "demographic surveys", such as monitoring the number of people who stop and examine various kiosks at a department store. These types of surveys can help you optimize the layout of a store and in turn make more sales. But even then you need to take *extreme care* to ensure that smaller groups of people are not marginalized. Keep in mind the law of large numbers as individual data points will average out in the population.

You should ***never*** use gender recognition to customize user experience (e.x., swapping gender-specific pronouns) based on (1) photos of individual users utilizing your software and (2) the output of your model. Not only are machine learning models far from perfect, you also have the moral responsibility to not assume a user's gender.

Age and gender classification is a very interesting task to examine from a computer vision perspective – not only is it challenging, but it's also a great example of a fine-grained classification task where subtle differences may make the difference in a correct or incorrect classification.

But therein lies the problem – ***people are not data points*** and ***gender is not binary***. There are very few ethical applications of gender recognition using artificial intelligence. Software that attempts to distill gender into binary classification only further chains us to antiquated notions of what gender is. Therefore, I would encourage you to *not* utilize gender recognition in your own applications if at all possible. If you *must* perform gender recognition, make sure you are holding yourself accountable and ensure you are not building applications that attempts to conform others to gender stereotypes.

Use this chapter to help you study computer vision and deep learning, but use your knowledge here for good. Please be respectful of your fellow humans and educate yourself on gender issues and gender equality.

My hope is I can one day remove this chapter from *Deep Learning for Computer Vision with Python*, not because people are using it for nefarious, egregious reasons, but rather that gender identification has become so antiquated that no one bothers to think about gender, let alone try to identify it.

14.2 The Adience Dataset

The Adience dataset [39] consists of 26,580 images of faces. A total of 2,284 subjects (people) were captured in this dataset. The dataset was then split into two gender classes (male and female) along with eight age groups; specifically: 0-2, 4-6, 8-13, 15-20, 25-32, 38-43, 48-53, and 60+. The goal of the Adience dataset is to correctly predict *both* the age and the gender of a given person in a photo. A sample of the dataset can be seen in Figure 14.1.

The dataset is imbalanced with more samples being present for people in the age brackets $25-32$ than than any other age group by a factor of two (4,897). Samples in the age ranges $48-53$ and $60+$ are the most unrepresented (825 and 869, respectively). The number of male samples (8,192) is slightly lower than the number of female data points (9,411), but is still within reasonable limits of class distribution.

When training a model on the Adience dataset, we have two choices:

1. Train a *single* classifier that is responsible for predicting both the gender and age classes, for a total of 16 possible classes.
2. Train *two* classifiers, one responsible for predicting gender (2 classes) and a separate model for age recognition (8 classes).

Both methods are viable options; however, we are going to opt for taking the approach of Levi et al. and train *two separate* CNNs – not only will doing so enable us to reproduce their results (and even outperform their reported accuracies), but it will enable our classifier to be more robust. The facial features learned to discriminate a $60+$ woman are likely extremely different than the features learned to recognize a man who is also $60+$; therefore, it makes sense to separate age and gender into two separate classifiers.

Figure 14.1: A sample of the Adience dataset for age and gender recognition. Age ranges span from $0-60+$.

14.2.1 Building the Adience Dataset

You can download the Adience dataset using the following link:

http://www.openu.ac.il/home/hassner/Adience/data.html

Simply navigate to the "Download" section and enter your name and email address (this information is only used to send you periodic emails if the dataset is ever updated). From there, you'll be able to login to the FTP site:

http://www.cslab.openu.ac.il/download/

You'll want to download the `aligned.tar.gz` file along with all `fold_frontal_*_data.txt` where the asterisk represents the digits 0-4 (Figure 14.2).

After downloading the dataset, you can unarchive it using the following command:

```
$ tar -xvf aligned.tar.gz
```

When organizing the dataset on my machine, I choose to use the following directory structure:

```
--- adience
|     |--- aligned
|     |--- folds
|     |--- lists
|     |--- rec
```

The `aligned` directory contains a number of subdirectories which contain the example faces we will use to train our age and gender CNNs. Strictly for organizational purposes, I also decided to store all fold `.txt` files (which Levi et al. originally used for cross-validation) inside a directory named `folds`. I then created the `lists` directory to store my generated `.lst` files along with a `rec` directory for the mxnet record databases.

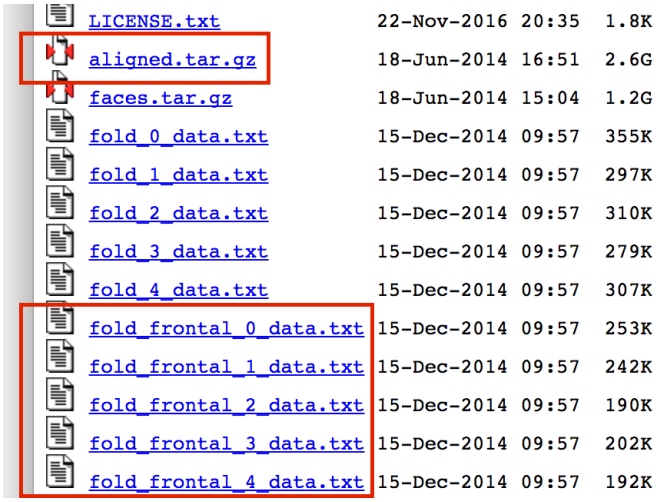


Figure 14.2: Make sure you download the `aligned.tar.gz` file along with all `fold_frontal_*_data.txt` files.

In the ImageNet and vehicle make and model chapters, we've computed both rank-1 and rank-5 accuracy; however, in context of Adience, rank-5 accuracy does not make intuitive sense. To start, computing rank-5 accuracy of a two class (gender) problem will always be (trivially) 100%. Therefore, when reporting the accuracy of the gender model, we'll simply report if a given classification is correct or not.

For the age model, we'll report what Levi et al. call "one-off" accuracy. One-off accuracy measures whether the *ground-truth* class label matches the *predicted* class label or if the ground-truth label exists in the two *adjacent bins*. For example, suppose our age model predicted the age bracket $15-20$; however, the ground-truth label is $25-32$ – according to the one-off evaluation metric, this value is still correct because $15-20$ exists in the set $\{15-20, 25-32, 38-48\}$.

Note that one-off accuracy is *not* the same thing as rank-2 accuracy. For example, if our model was presented with an image containing a person of age $48-53$ (i.e., the ground-truth label) and our model predicted:

- $4-6$ with probability 63.7%
- $48-53$ with probability 36.3%

Then we would consider this an *incorrect* classification since the first predicted label $(4-6)$ is not *adjacent* to the $48-53$ bin. If instead our model predicted:

- $38-43$ with probability 63.7%
- $48-53$ with probably 36.3%

Then this *would* be a correct one-off prediction since the $38-43$ bin is *directly adjacent* to the $48-53$ bin. We will learn how to implement custom evaluation metrics such as one-off accuracy later in this chapter.

The Adience Configuration File

Just as in our ImageNet experiments, we need to first define a directory structure of our project:

```
|--- age_gender
|    |--- config
|    |    |--- __init__.py
|    |    |--- age_gender_config.py
|    |    |--- age_gender_deploy.py
```

```
|    |--- checkpoints
|    |    |--- age/
|    |    |--- gender/
|    |--- build_dataset.py
|    |--- test_accuracy.py
|    |--- test_prediction.py
|    |--- train.py
|    |--- vis_classification.py
```

Inside the `config` sub-module, we'll create two separate configuration files. One file `age_gender_config.py` will be used when we are *training* our CNNs. The second file, `age_gender_deploy.py` will be used after training is complete, and we wish to apply our CNNs to images *outside* the Adience dataset.

The `checkpoints` directory will store all model checkpoints during training. The `age` subdirectory will contain all checkpoints related to the age CNN while the `gender` subdirectory will store all checkpoints related to the gender CNN.

The `build_dataset.py` script will be responsible for creating our `.lst` and `.rec` files for both the age and gender splits. Once we have created our datasets, we can use `train.py` to train our CNNs. To evaluate the performance of our networks on the testing set, we'll use `test_accuracy.py`. We'll then be able to visualize predictions from *within* the Adience dataset using `vis_classification.py`. Anytime we want to classify an image *outside* of Adience, we'll use `test_prediction.py`.

Let's go ahead and review the contents of `age_gender_config.py` now:

```
1  # import the necessary packages
2  from os import path
3
4  # define the type of dataset we are training (i.e., either "age" or
5  # "gender")
6  DATASET_TYPE = "gender"
7
8  # define the base paths to the faces dataset and output path
9  BASE_PATH = "/raid/datasets/adience"
10 OUTPUT_BASE = "output"
11 MX_OUTPUT = BASE_PATH
12
13 # based on the base path, derive the images path and folds path
14 IMAGES_PATH = path.sep.join([BASE_PATH, "aligned"])
15 LABELS_PATH = path.sep.join([BASE_PATH, "folds"])
```

On **Line 6** we define the `DATASET_TYPE`. This value can either be `gender` or `age`. Depending on the `DATASET_TYPE` we will change various output paths and settings later in this configuration file.

We then define the `BASE_PATH` to the audience dataset on **Line 9** using the directory structure I detailed above – you will need to change the `BASE_PATH` to point to where your Adience dataset lives on disk. **Line 14** uses the `BASE_PATH` to construct the path to our `aligned` directory which contains the input images. **Line 15** also utilizes `BASE_PATH` to build the path to the `folds` directory where we store our `.txt` files containing the class labels.

Next, let's define some information on our training, testing, and validation splits:

```
17 # define the percentage of validation and testing images relative
18 # to the number of training images
19 NUM_VAL_IMAGES = 0.15
20 NUM_TEST_IMAGES = 0.15
```

We'll use 70% of our data for training. The remaining 30% will be equally split, 15% for validation and 15% for testing. We'll also train our network using batch sizes of 128 and two GPUs:

```
22 # define the batch size
23 BATCH_SIZE = 128
24 NUM_DEVICES = 2
```

I used two GPUs on this experiment to enable me to quickly gather results (in the order of 17 second epochs). You can *easily* train your models using a single GPU as well.

Our next code block handles if we are training a CNN on the "age" dataset:

```
26 # check to see if we are working with the "age" portion of the
27 # dataset
28 if DATASET_TYPE == "age":
29     # define the number of labels for the "age" dataset, along with
30     # the path to the label encoder
31     NUM_CLASSES = 8
32     LABEL_ENCODER_PATH = path.sep.join([OUTPUT_BASE,
33         "age_le.cpickle"])
34
35     # define the path to the output training, validation, and testing
36     # lists
37     TRAIN_MX_LIST = path.sep.join([MX_OUTPUT, "lists/age_train.lst"])
38     VAL_MX_LIST = path.sep.join([MX_OUTPUT, "lists/age_val.lst"])
39     TEST_MX_LIST = path.sep.join([MX_OUTPUT, "lists/age_test.lst"])
40
41     # define the path to the output training, validation, and testing
42     # image records
43     TRAIN_MX_REC = path.sep.join([MX_OUTPUT, "rec/age_train.rec"])
44     VAL_MX_REC = path.sep.join([MX_OUTPUT, "rec/age_val.rec"])
45     TEST_MX_REC = path.sep.join([MX_OUTPUT, "rec/age_test.rec"])
46
47     # derive the path to the mean pixel file
48     DATASET_MEAN = path.sep.join([OUTPUT_BASE,
49         "age_adience_mean.json"])
```

Line 31 defines the `NUM_CLASSES` for the age dataset, which is eight overall age brackets. We'll also explicitly define the `LABEL_ENCODER_PATH` which will store the serialized `LabelEncoder` object.

Lines 37-39 define the paths to the `.lst` files while **Lines 43-45** do the same, only for the `.rec` files. We'll also want to make sure we store the RGB means for the training set by defining the `DATASET_MEAN` (**Lines 48 and 49**).

In the case that we are instead training a CNN on the gender data, we'll initialize the *same* set of variables, only with different image paths:

```
51 # otherwise, check to see if we are performing "gender"
52 # classification
```

```
elif DATASET_TYPE == "gender":
	# define the number of labels for the "gender" dataset, along
	# with the path to the label encoder
	NUM_CLASSES = 2
	LABEL_ENCODER_PATH = path.sep.join([OUTPUT_BASE,
		"gender_le.cpickle"])

	# define the path to the output training, validation, and testing
	# lists
	TRAIN_MX_LIST = path.sep.join([MX_OUTPUT,
		"lists/gender_train.lst"])
	VAL_MX_LIST = path.sep.join([MX_OUTPUT,
		"lists/gender_val.lst"])
	TEST_MX_LIST = path.sep.join([MX_OUTPUT,
		"lists/gender_test.lst"])

	# define the path to the output training, validation, and testing
	# image records
	TRAIN_MX_REC = path.sep.join([MX_OUTPUT, "rec/gender_train.rec"])
	VAL_MX_REC = path.sep.join([MX_OUTPUT, "rec/gender_val.rec"])
	TEST_MX_REC = path.sep.join([MX_OUTPUT, "rec/gender_test.rec"])

	# derive the path to the mean pixel file
	DATASET_MEAN = path.sep.join([OUTPUT_BASE,
		"gender_adience_mean.json"])
```

Notice now the `NUM_CLASSES` variable has been changed to two (either "male" or "female"), followed by replacing all occurrences of "age" with "gender" in the file paths. Now that our configuration file is created, let's move on to building the helper class used to facilitate building the dataset and converting it to a format we can use with mxnet.

The Adience Helper Class

In order to work with the Adience dataset, we'll first need to define a set of utility functions, similar what we did with the ImageNet dataset. This function will be responsible for helping us build our class labels, image paths, and aiding in computing accuracy. Let's name this file `agegenderhelper.py` and include it in the `pyimagesearch.utils` sub-module:

```
| --- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- io
|    |--- nn
|    |--- preprocessing
|    |--- utils
|    |    |--- __init__.py
|    |    |--- agegenderhelper.py
|    |    |--- captchahelper.py
|    |    |--- imagenethelper.py
|    |    |--- ranked.py
```

When you're ready, open up `agegenderhelper.py`, and we'll get to work:

```
1  # import the necessary packages
2  import numpy as np
3  import glob
4  import cv2
5  import os
6
7  class AgeGenderHelper:
8       def __init__(self, config):
9            # store the configuration object and build the age bins used
10           # for constructing class labels
11           self.config = config
12           self.ageBins = self.buildAgeBins()
```

Lines 2-5 import our required Python packages while **Line 8** defines the constructor to our `AgeGenderHelper` class. The constructor requires a single argument, `config` which is assumed to be the `age_gender_config` module. We store this `config` object on **Line 11** and build the set of `ageBins` on **Line 12**.

We define the `buildAgeBins` function below:

```
14      def buildAgeBins(self):
15           # initialize the list of age bins based on the Adience
16           # dataset
17           ageBins = [(0, 2), (4, 6), (8, 13), (15, 20), (25, 32),
18                (38, 43), (48, 53), (60, np.inf)]
19
20           # return the age bins
21           return ageBins
```

Lines 17 and 18 create a list of 2-tuples that represent the *lower boundary* on the age bracket and the *upper boundary* on the age bracket. As you can see, there are eight age brackets defined, identical to Levi et al. The `ageBins` are returned to the calling function on **Line 21**.

Given an arbitrary age bracket or gender name, we need to encode the input as a label – the following function handles this:

```
23      def toLabel(self, age, gender):
24           # check to see if we should determine the age label
25           if self.config.DATASET_TYPE == "age":
26                return self.toAgeLabel(age)
27
28           # otherwise, assume we are determining the gender label
29           return self.toGenderLabel(gender)
```

This function accepts both an `age` and `gender`. If we are working with the age dataset, then we call `toAgeLabel` and return the value. Otherwise, we'll assume we are working with the gender dataset and encode the gender label instead.

The next step is to define the `toAgeLabel` function:

```
31      def toAgeLabel(self, age):
32           # initialize the label
33           label = None
34
```

```
35          # break the age tuple into integers
36          age = age.replace("(", "").replace(")", "").split(", ")
37          (ageLower, ageUpper) = np.array(age, dtype="int")
38
39          # loop over the age bins
40          for (lower, upper) in self.ageBins:
41               # determine if the age falls into the current bin
42               if ageLower >= lower and ageUpper <= upper:
43                    label = "{}_{}".format(lower, upper)
44                    break
45
46          # return the label
47          return label
```

Here we pass in a single parameter, the `age` string. Inside the Adience dataset, age labels are represented as strings in the form `(0, 2)`, `(4, 6)`, etc. In order to determine the correct label for this input string, we first need to extract the lower and upper age boundaries as integers (**Lines 36 and 37**).

We then loop over our `ageBins` (**Line 40**) and determine which bin the supplied `age` falls into (**Line 42**). Once we have found the correct bin, we build our label as a string with the lower and upper boundaries separated by an underscore. For example, if an input `age` of `(8, 13)` was passed into this function, the output would be `8_13`. Performing this encoding makes it easier for us to parse the labels, both when training and evaluating our network.

Now that we can create age labels, we need to construct gender labels as well:

```
49     def toGenderLabel(self, gender):
50          # return 0 if the gender is male, 1 if the gender is female
51          return 0 if gender == "m" else 1
```

This function is straightforward – if the input `gender` is `m` (for "male") we return `0`; otherwise, the gender is assumed to be female, so we return `1`.

When evaluating one-off accuracy, we need an efficient, fast method to determine if the *predicted* class label is *equal to* or *adjacent to* the ground-truth label. The easiest way to accomplish this task is to define a dictionary that maps a ground-truth label to its corresponding adjacent labels. For example, if we knew the ground-truth label to a given data point is `8_13`, we could use this value as a key to our dictionary. The value would then be `["4_6", "8_13", "15_20"]` – the set of adjacent labels. Simply checking if `8_13` exists in this list would enable us to quickly evaluate one-off accuracy.

To define this dictionary, we'll create a more complex function, `buildOneOffMappings`:

```
53     def buildOneOffMappings(self, le):
54          # sort the class labels in ascending order (according to age)
55          # and initialize the one-off mappings for computing accuracy
56          classes = sorted(le.classes_, key=lambda x:
57               int(x.decode("utf-8").split("_")[0]))
58          oneOff = {}
```

Our method will accept a single required argument, `le`, which is an instantiation of the `LabelEncoder` object used to encode age class labels. As we know from earlier in the definition of this class, age labels will be encoded in the form {lower}_{upper}. To determine adjacent age

boundaries, we first need to extract the class label names from `le` and then sort label names in *ascending* order based on the lower boundary (**Lines 56 and 57**). **Line 58** defines the `oneOff` dictionary we will use to quickly evaluate one-off accuracy.

Next, we need to loop over the index and name of the sorted class labels:

```
60          # loop over the index and name of the (sorted) class labels
61          for (i, name) in enumerate(classes):
62               # determine the index of the *current* class label name
63               # in the *label encoder* (unordered) list, then
64               # initialize the index of the previous and next age
65               # groups adjacent to the current label
66               current = np.where(le.classes_ == name)[0][0]
67               prev = -1
68               next = -1
69
70               # check to see if we should compute previous adjacent
71               # age group
72               if i > 0:
73                    prev = np.where(le.classes_ == classes[i - 1])[0][0]
74
75               # check to see if we should compute the next adjacent
76               # age group
77               if i < len(classes) - 1:
78                    next = np.where(le.classes_ == classes[i + 1])[0][0]
79
80               # construct a tuple that consists of the current age
81               # bracket, the previous age bracket, and the next age
82               # bracket
83               oneOff[current] = (current, prev, next)
84
85          # return the one-off mappings
86          return oneOff
```

Line 66 determines the index of the `current` class label name. Provided that our age bracket is greater than the $0-2$ range, we then determine the previous adjacent bracket (**Lines 72 and 73**). In the case that the `current` age bracket is less than the $60+$ range, we then determine the next adjacent bracket (**Lines 77 and 78**).

The `oneOff` dictionary is then updated using the `current` age bracket as a key with a 3-tuple consisting of the `current` label, along with the two adjacent `prev` and `next` class labels. The `oneOff` dictionary is then returned to the calling function on **Line 86**.

Utility functions such as these are great examples of why it's important to learn the fundamentals of the Python programming, the NumPy library, and have at least a basic understanding of the scikit-learn library. Using these techniques, we can easily build methods that enable us to evaluate complex metrics efficiently. If you find yourself having trouble understanding this function, take the time to work with it and insert `print` statements so you can see how a given age class in the `le` label encoder is able to have its two adjacent age brackets computed.

Our next function handles building the image paths and corresponding labels to the data points in the Adience dataset:

```
88      def buildPathsAndLabels(self):
89          # initialize the list of image paths and labels
90          paths = []
```

```
91          labels = []
92
93          # grab the paths to the folds files
94          foldPaths = os.path.sep.join([self.config.LABELS_PATH,
95               "*.txt"])
96          foldPaths = glob.glob(foldPaths)
```

On **Lines 90 and 91** we initialize our `paths` and `labels` lists, respectively. Both the age and gender of a given person are stored inside the `LABELS_PATH` files or the "fold" `.txt` files – we grab the paths to all fold files on **Lines 94-96**.

Next, let's loop over each of the `foldPaths` and parse them individually:

```
98          # loop over the folds paths
99          for foldPath in foldPaths:
100              # load the contents of the folds file, skipping the
101              # header
102              rows = open(foldPath).read()
103              rows = rows.strip().split("\n")[1:]
104
105              # loop over the rows
106              for row in rows:
107                  # unpack the needed components of the row
108                  row = row.split("\t")
109                  (userID, imagePath, faceID, age, gender) = row[:5]
110
111                  # if the age or gender is invalid, ignore the sample
112                  if age[0] != "(" or gender not in ("m", "f"):
113                      continue
```

For all of the `foldPath` files, we load the contents on **Lines 102 and 103**. We then loop over each of the `rows` in the input file on **Line 106**. Each row is tab separated, which we break into a list of strings on **Line 108**. We then extract the five first entries from the row:

1. `userID`: Unique ID of the subject in the photo.
2. `imagePath`: The path to the input image.
3. `faceID`: Unique ID of the face itself.
4. `age`: The age label encoded in a string in the format `(lower, upper)`.
5. `gender`: The gender label represented as a single character, either `m` or `f`.

Lines 112 and 113 ensure that the data point is valid by ensuring the `age` is properly encoded and the gender is either male or female. If either of these conditions do not hold, we throw out the data point due to ambiguous labeling. Provided the labels pass our sanity checks, we can move on to building our input image paths and encoding the label:

```
115                  # construct the path to the input image and build
116                  # the class label
117                  p = "landmark_aligned_face.{}.{}".format(faceID,
118                      imagePath)
119                  p = os.path.sep.join([self.config.IMAGES_PATH,
120                      userID, p])
121                  label = self.toLabel(age, gender)
```

Lines 117 and 118 start building the path to the input image by combining the `faceID` with the filename. We then finish constructing the path to the image on **Lines 119 and 120**.

We can validate this code works by inspecting an example image path in the Adience dataset:

```
adience/aligned/100003415@N08/landmark_aligned_face.2174.9523333835_c7887c3fde_o.jpg
```

Notice how the first component of the path is `adience`, our base directory. The next sub-directory is the `userID`. Inside the `userID` sub-directory, there are a number of images, each starting with the base filename `landmark_aligned_face.*`. To finish building the image path, we supply the `faceID`, followed by the rest of the filename, `imagePath`. Again, you can validate this code yourself by examining the file paths in the Adience dataset. **Line 121** then encodes the `label` using our `toLabel` function.

Finally, we can update our `paths` and `labels` lists:

```
123                     # if the label is None, then the age does not fit
124                     # into our age brackets, ignore the sample
125                     if label is None:
126                         continue
127
128                     # update the respective image paths and labels lists
129                     paths.append(p)
130                     labels.append(label)
131
132             # return a tuple of image paths and labels
133             return (paths, labels)
```

Line 125 handles the case when our `label` is `None`, which occurs when the `age` does not fit into our age brackets, likely due to either (1) an error when hand labeling the dataset or (2) a deprecated age bin that was not removed from the Adience dataset. In either case, we discard the data point since the `label` is ambiguous. **Lines 129 and 130** update our `paths` and `labels` lists, respectively, which are then returned to the calling method on **Line 133**.

Two more functions are defined in our `AgeGenderHelper` class named `visualizeAge` and `visualizeGender`, respectively. These functions accept a predicted probability distribution for each label and construct a simple bar chart enabling us to visualize the class label distribution, *exactly* as we did in Chapter 11 on emotion and facial expression recognition. As this is already a lengthy chapter, I will leave reviewing these functions as an exercise to the reader – they are simply used for visualization and play no part in the role of building a deep learning classifier to recognize age and gender.

Creating the List Files

Our `build_dataset.py` script is near identical to the script we used when building the vehicle make and model dataset in Chapter 13. Let's go ahead and (briefly) review the file name:

```
1  # import the necessary packages
2  from config import age_gender_config as config
3  from sklearn.preprocessing import LabelEncoder
4  from sklearn.model_selection import train_test_split
5  from pyimagesearch.utils import AgeGenderHelper
6  import numpy as np
7  import progressbar
8  import pickle
9  import json
```

```
10 import cv2
11
12 # initialize our helper class, then build the set of image paths
13 # and class labels
14 print("[INFO] building paths and labels...")
15 agh = AgeGenderHelper(config)
16 (trainPaths, trainLabels) = agh.buildPathsAndLabels()
```

Lines 2-10 import our required Python packages. Note the import of our `age_gender_config` – this import will allow us to use the *same* `build_dataset.py` script, regardless if we are building the *age* or *gender* dataset. To facilitate our ability to build the input image paths and corresponding class labels based on the `DATASET_TYPE`, we'll also import our newly implemented `AgeGenderHelper`. Using the `AgeGenderHelper`, we then build the paths to the input images and class labels on **Lines 15 and 16**.

Now that we have the total number of images in the training set, let's derive the number of images for the validation and testing set, respectively:

```
18 # now that we have the total number of images in the dataset that
19 # can be used for training, compute the number of images that
20 # should be used for validation and testing
21 numVal = int(len(trainPaths) * config.NUM_VAL_IMAGES)
22 numTest = int(len(trainPaths) * config.NUM_TEST_IMAGES)
23
24 # our class labels are represented as strings so we need to encode
25 # them
26 print("[INFO] encoding labels...")
27 le = LabelEncoder().fit(trainLabels)
28 trainLabels = le.transform(trainLabels)
```

Lines 27 and 28 encode our input labels from strings to integers. Next, let's sample `numVal` validation images from the training set:

```
30 # perform sampling from the training set to construct a a validation
31 # set
32 print("[INFO] constructing validation data...")
33 split = train_test_split(trainPaths, trainLabels, test_size=numVal,
34     stratify=trainLabels)
35 (trainPaths, valPaths, trainLabels, valLabels) = split
```

The same is then done for `numTest` images to create our testing set:

```
37 # perform stratified sampling from the training set to construct a
38 # a testing set
39 print("[INFO] constructing testing data...")
40 split = train_test_split(trainPaths, trainLabels, test_size=numTest,
41     stratify=trainLabels)
42 (trainPaths, testPaths, trainLabels, testLabels) = split
```

We'll then build our `datasets` lists, where each entry is a 4-tuple containing the dataset split type, image paths, class labels, and output `.lst` file:

```
# construct a list pairing the training, validation, and testing
# image paths along with their corresponding labels and output list
# files
datasets = [
     ("train", trainPaths, trainLabels, config.TRAIN_MX_LIST),
     ("val", valPaths, valLabels, config.VAL_MX_LIST),
     ("test", testPaths, testLabels, config.TEST_MX_LIST)]

# initialize the lists of RGB channel averages
(R, G, B) = ([], [], [])
```

We'll also initialize our RGB mean values as well – these values will enable us to perform mean normalization during the training process.

Next, we need to loop over each of the entries in the `datasets` list:

```
# loop over the dataset tuples
for (dType, paths, labels, outputPath) in datasets:
     # open the output file for writing
     print("[INFO] building {}...".format(outputPath))
     f = open(outputPath, "w")

     # initialize the progress bar
     widgets = ["Building List: ", progressbar.Percentage(), " ",
          progressbar.Bar(), " ", progressbar.ETA()]
     pbar = progressbar.ProgressBar(maxval=len(paths),
          widgets=widgets).start()
```

Line 59 opens a file pointer to the current `.lst` file. We also initialize `progressbar` widgets on **Lines 62-65**. The progress bar isn't a requirement, but it is often nice to have ETA information displayed to our screen when building a dataset.

Our next code block handles looping over each of the image paths and corresponding labels in the data split:

```
     # loop over each of the individual images + labels
     for (i, (path, label)) in enumerate(zip(paths, labels)):
          # if we are building the training dataset, then compute the
          # mean of each channel in the image, then update the
          # respective lists
          if dType == "train":
               image = cv2.imread(path)
               (b, g, r) = cv2.mean(image)[:3]
               R.append(r)
               G.append(g)
               B.append(b)

          # write the image index, label, and output path to file
          row = "\t".join([str(i), str(label), path])
          f.write("{}\n".format(row))
          pbar.update(i)

     # close the output file
     pbar.finish()
     f.close()
```

If we are examining the training split, we load the image from disk, compute the RGB mean, and update our average lists (**Lines 72-77**). Otherwise, we write an mxnet-formatted row to the output file containing the unique image integer `i`, the class `label`, and the `path` to the input image (**Lines 80 and 81**). **Line 86** closes the file pointer and the `for` loop restarts for the next data split (until we have created `.lst` files for each of the data splits).

Our final code block handles serializing the RGB means to disk, as well as the label encoder:

```
88  # construct a dictionary of averages, then serialize the means to a
89  # JSON file
90  print("[INFO] serializing means...")
91  D = {"R": np.mean(R), "G": np.mean(G), "B": np.mean(B)}
92  f = open(config.DATASET_MEAN, "w")
93  f.write(json.dumps(D))
94  f.close()
95
96  # serialize the label encoder
97  print("[INFO] serializing label encoder...")
98  f = open(config.LABEL_ENCODER_PATH, "wb")
99  f.write(pickle.dumps(le))
100 f.close()
```

To construct our `.lst` files for the "gender" dataset, ensure that `DATASET_TYPE` is set to `gender` in `age_gender_config.py` and execute the following command:

```
$ python build_dataset.py
[INFO] building paths and labels...
[INFO] encoding labels...
[INFO] constructing validation data...
[INFO] constructing testing data...
[INFO] building /raid/datasets/adience/lists/gender_train.lst...
Building List: 100% |####################################| Time: 0:01:01
[INFO] building /raid/datasets/adience/lists/gender_val.lst...
Building List: 100% |####################################| Time: 0:00:00
[INFO] building /raid/datasets/adience/lists/gender_test.lst...
Building List: 100% |####################################| Time: 0:00:00
[INFO] serializing means...
[INFO] serializing label encoder...
```

You can validate that the gender `.lst` files were properly created by listing the contents of your `*_MX_LIST` variables:

```
$ wc -l adience/lists/gender_*.lst
   1712 adience/lists/gender_test.lst
   7991 adience/lists/gender_train.lst
   1712 adience/lists/gender_val.lst
  11415 total
```

Here you can see that we have a total of 11,415 images in our dataset, with $\approx 8,000$ images for training and $\approx 1,700$ for both validation and testing.

To create the `.lst` files for the "age" dataset, go back to the `age_gender_config.py` file and update `DATASET_TYPE` to be `age`. Then, once again execute `build_dataset.py`:

```
$ python build_dataset.py
[INFO] building paths and labels...
[INFO] encoding labels...
[INFO] constructing validation data...
[INFO] constructing testing data...
[INFO] building /raid/datasets/adience/lists/age_train.lst...
Building List: 100% |####################################| Time: 0:00:52
[INFO] building /raid/datasets/adience/lists/age_val.lst...
Building List: 100% |####################################| Time: 0:00:00
[INFO] building /raid/datasets/adience/lists/age_test.lst...
Building List: 100% |####################################| Time: 0:00:00
[INFO] serializing means...
[INFO] serializing label encoder...
```

Once again, validate that your `.lst` files have been successfully created for each of the data splits:

```
$ wc -l adience/lists/age_*.lst
   1711 adience/lists/age_test.lst
   7986 adience/lists/age_train.lst
   1711 adience/lists/age_val.lst
  11408 total
```

Creating the Record Database

Given our `.lst` files for the age and gender datasets, let's create our `.rec` files using mxnet's `im2rec` binary. We'll start by creating the age record files:

```
$ ~/mxnet/bin/im2rec /raid/datasets/adience/lists/age_train.lst "" \
      /raid/datasets/adience/rec/age_train.rec resize=256 encoding='.jpg' \
      quality=100
...
[07:28:16] tools/im2rec.cc:298: Total: 7986 images processed, 42.5361 sec elapsed
$ ~/mxnet/bin/im2rec /raid/datasets/adience/lists/age_val.lst "" \
     /raid/datasets/adience/rec/age_val.rec resize=256 encoding='.jpg' \
     quality=100
...
[07:28:51] tools/im2rec.cc:298: Total: 1711 images processed, 10.5159 sec elapsed
$ ~/mxnet/bin/im2rec /raid/datasets/adience/lists/age_test.lst "" \
     /raid/datasets/adience/rec/age_test.rec resize=256 encoding='.jpg' \
     quality=100
...
[07:29:20] tools/im2rec.cc:298: Total: 1711 images processed, 10.7158 sec elapsed
```

And then switch over to the gender record files:

```
$ ~/mxnet/bin/im2rec /raid/datasets/adience/lists/gender_train.lst "" \
     /raid/datasets/adience/rec/gender_train.rec resize=256 encoding='.jpg' \
     quality=100
...
[07:30:35] tools/im2rec.cc:298: Total: 7991 images processed, 43.2748 sec elapsed
$ ~/mxnet/bin/im2rec /raid/datasets/adience/lists/gender_val.lst "" \
     /raid/datasets/adience/rec/gender_val.rec resize=256 encoding='.jpg' \
     quality=100
```

```
...
[07:31:00] tools/im2rec.cc:298: Total: 1712 images processed, 9.81215 sec elapsed
$ ~/mxnet/bin/im2rec /raid/datasets/adience/lists/gender_test.lst "" \
     /raid/datasets/adience/rec/gender_test.rec resize=256 encoding='.jpg' \
     quality=100
...
[07:31:27] tools/im2rec.cc:298: Total: 1712 images processed, 9.5392 sec elapsed
```

To validate that your record files were created, simply check the paths of your `*_MX_REC` variables:

```
$ ls -l adience/rec/
total 1082888
-rw-rw-r-- 1 adrian adrian  82787512 Aug 28 07:29 age_test.rec
-rw-rw-r-- 1 adrian adrian 387603688 Aug 28 07:28 age_train.rec
-rw-rw-r-- 1 adrian adrian  83883944 Aug 28 07:28 age_val.rec
-rw-rw-r-- 1 adrian adrian  83081304 Aug 28 07:31 gender_test.rec
-rw-rw-r-- 1 adrian adrian 388022840 Aug 28 07:30 gender_train.rec
-rw-rw-r-- 1 adrian adrian  83485564 Aug 28 07:31 gender_val.rec
```

Sure enough, all six of our files are there: Three for the training, testing, and validation split for *age*. And three files for the training testing, and validation split for *gender*. Each of the training files are approximately 388MB, while all of the testing and validation files weigh in at $\approx 83MB$. We'll be using these record files to train our CNN – but first, let's define the network architecture itself.

14.3 Implementing Our Network Architecture

The network architecture used by Levi et al. is similar to AlexNet (Chapter 6), only:

1. More shallow with no multiple `CONV => RELU` layers stacked on top of each other.
2. Fewer nodes in the fully-connected layers.

We will replicate their exact architecture, with two exceptions. First, we will utilize batch normalization rather than the now deprecated Local Response Normalization (LRN) used in earlier CNN architectures. Secondly, we'll introduce a small amount of dropout after every pooling layer to help reduce overfitting.

Let's go ahead and implement the Levi et al. architecture, which we will call `MxAgegenderNet`. Create a new file named `mxagegendernet.py` inside the `nn.mxconv` sub-module of `pyimagesearch`:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- nn
|    |    |--- __init__.py
|    |    |--- conv
|    |    |--- mxconv
|    |    |    |--- __init__.py
|    |    |    |--- mxagegender.py
|    |    |    |--- mxalexnet.py
|    |    |    |--- mxgooglenet.py
|    |    |    |--- mxresnet.py
|    |    |    |--- mxsqueezenet.py
|    |    |    |--- mxvggnet.py
|    |--- preprocessing
|    |--- utils
```

From there, open up the file and insert the following code:

```
1  # import the necessary packages
2  import mxnet as mx
3
4  class MxAgeGenderNet:
5      @staticmethod
6      def build(classes):
7          # data input
8          data = mx.sym.Variable("data")
9
10         # Block #1: first CONV => RELU => POOL layer set
11         conv1_1 = mx.sym.Convolution(data=data, kernel=(7, 7),
12             stride=(4, 4), num_filter=96)
13         act1_1 = mx.sym.Activation(data=conv1_1, act_type="relu")
14         bn1_1 = mx.sym.BatchNorm(data=act1_1)
15         pool1 = mx.sym.Pooling(data=bn1_1, pool_type="max",
16             kernel=(3, 3), stride=(2, 2))
17         do1 = mx.sym.Dropout(data=pool1, p=0.25)
```

Line 6 defines the `build` method of our network, as in all previous architectures covered in this book. The `data` variable on **Line 8** serves as the input to our network.

From there, we define a series of `CONV => RELU => POOL` layers on **Lines 11-17**. Our first `CONV` layer in the network will learn 96, 7×7 kernels with a stride of 4×4 used to reduce the spatial dimensions of the input 227×227 images. An activation is applied after the convolution, followed by a batch normalization, following the recommended usage of batch normalization (Chapter 11, *Starter Bundle*).

Max pooling is applied on **Lines 15 and 16** with a kernel size of 3×3 and stride of 2×2 to once again reduce spatial dimensions. Dropout with a probability of 25% is used to help reduce overfitting.

Our second series of layers applies the same structure, only this time adjusting the `CONV` layer to learn 256, 5×5 filters:

```
19         # Block #2: second CONV => RELU => POOL layer set
20         conv2_1 = mx.sym.Convolution(data=do1, kernel=(5, 5),
21             pad=(2, 2), num_filter=256)
22         act2_1 = mx.sym.Activation(data=conv2_1, act_type="relu")
23         bn2_1 = mx.sym.BatchNorm(data=act2_1)
24         pool2 = mx.sym.Pooling(data=bn2_1, pool_type="max",
25             kernel=(3, 3), stride=(2, 2))
26         do2 = mx.sym.Dropout(data=pool2, p=0.25)
```

The final `CONV => RELU => POOL` layer set is near identical, only the number of filters is increased to 384 and the filter size reduced to 3×3:

```
28         # Block #3: second CONV => RELU => POOL layer set
29         conv2_1 = mx.sym.Convolution(data=do2, kernel=(3, 3),
30             pad=(1, 1), num_filter=384)
31         act2_1 = mx.sym.Activation(data=conv2_1, act_type="relu")
32         bn2_1 = mx.sym.BatchNorm(data=act2_1)
33         pool2 = mx.sym.Pooling(data=bn2_1, pool_type="max",
```

```
34                 kernel=(3, 3), stride=(2, 2))
35             do3 = mx.sym.Dropout(data=pool2, p=0.25)
```

Next comes our first set of fully-connected layers:

```
37             # Block #4: first set of FC => RELU layers
38             flatten = mx.sym.Flatten(data=do3)
39             fc1 = mx.sym.FullyConnected(data=flatten, num_hidden=512)
40             act4_1 = mx.sym.Activation(data=fc1, act_type="relu")
41             bn4_1 = mx.sym.BatchNorm(data=act4_1)
42             do4 = mx.sym.Dropout(data=bn4_1, p=0.5)
```

Unlike the AlexNet architecture where 4,096 hidden nodes are learned, we only learn 512 nodes here. We also apply an activation followed by a batch normalization in this layer set as well.

The second set of fully-connected layers is then applied:

```
44             # Block #5: second set of FC => RELU layers
45             fc2 = mx.sym.FullyConnected(data=do4, num_hidden=512)
46             act5_1 = mx.sym.Activation(data=fc2, act_type="relu")
47             bn5_1 = mx.sym.BatchNorm(data=act5_1)
48             do5 = mx.sym.Dropout(data=bn5_1, p=0.5)
```

Our final code block is simply an `FC` layer to learn the desired number of `classes`, along with a softmax classifier:

```
50             # softmax classifier
51             fc3 = mx.sym.FullyConnected(data=do5, num_hidden=classes)
52             model = mx.sym.SoftmaxOutput(data=fc3, name="softmax")
53
54             # return the network architecture
55             return model
```

In terms of network architectures, `MxAgeGenderNet` is arguably the simplest network architecture we have examined during the entirety of the *ImageNet Bundle*. However, as our results will demonstrate, this simple, sequential network architecture is not only able to replicate the results of Levi et al., but *improve* on their work as well.

14.4 Measuring "One-off" Accuracy

As mentioned in Section 14.2.1 above, when training and evaluating MxAgeGenderNet on the *gender* dataset, we'll want to compute "one-off accuracy". Unlike rank-5 accuracy, this metric marks a given prediction as "correct" if the predicted label is either (1) the ground-truth label or (2) *directly adjacent* to the ground-truth label.

In practice, this evaluation metric makes practical sense, and discriminating age is often highly subjective and highly correlated with genetics, appearances, and environmental factors. For example, an 18 year old man who smokes a carton of cigarettes a day for 20 years will likely look much older at 38 than the biologically correct 38 year old man.

To create our special one-off metric, let's create a new sub-module inside `pyimagesearch` named `mxcallbacks`, and inside this new sub-module, we'll place a new file – `mxmetrics.py`:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- mxcallbacks
|    |    |--- __init__.py
|    |    |--- mxmetrics.py
|    |--- nn
|    |--- preprocessing
|    |--- utils
```

The `mxmetrics.py` function will store our function to compute the one-off evaluation metric. Defining mxnet callback metrics is unfortunately not as easy as it is with Keras, so we'll need to take some special care here defining the function. Open up `mxmetrics.py` and we'll get to work:

```
1  # import the necessary packages
2  import mxnet as mx
3  import logging
4
5  def one_off_callback(trainIter, testIter, oneOff, ctx):
6      def _callback(iterNum, sym, arg, aux):
7          # construct a model for the symbol so we can make predictions
8          # on our data
9          model = mx.mod.Module(symbol=sym, context=ctx)
10         model.bind(data_shapes=testIter.provide_data,
11             label_shapes=testIter.provide_label)
12         model.set_params(arg, aux)
13
14         # compute one-off metric for both the training and testing
15         # data
16         trainMAE = _compute_one_off(model, trainIter, oneOff)
17         testMAE = _compute_one_off(model, testIter, oneOff)
18
19         # log the values
20         logging.info("Epoch[{}] Train-one-off={:.5f}".format(iterNum,
21             trainMAE))
22         logging.info("Epoch[{}] Test-one-off={:.5f}".format(iterNum,
23             testMAE))
24
25     # return the callback method
26     return _callback
```

Lines 2 and 3 import our required Python packages. We'll need the `logging` package in order to log the one-off evaluation metric to the same log file as our training progress.

From there, we define the actual callback function, `one_off_callback` on **Line 5**. This function accepts four required parameters:

- `trainIter`: The `ImageRecordIter` for the training set.
- `testIter`: The `imageRecordIter` for either the validation or testing set.
- `oneOff`: The dictionary of one-off mappings constructed by the `AgeGenderHelper` class.
- `ctx`: The context (i.e., CPU/GPU/etc.) that will be used during the evaluation.

In order to turn this into a true "callback" function, mxnet requires us to define an encapsulated Python function (i.e., an "inner" function) named `_callback` – those unfamiliar with encapsulated functions should first read this great introduction (http://pyimg.co/fbchc). Per the mxnet documentation, the encapsulated `_callback` function must accept four parameters:

1. The iteration (i.e., epoch) number of the network that is being trained.
2. The network symbol (`sym`) which represents the model weights after the current epoch.
3. The argument parameters (`arg`) to the network.
4. The auxiliary parameters (`aux`).

Based on these values, we can create a `Module` for the network parameters on **Lines 9-12**. We then call `_compute_one_off` (to be defined in the next section) on **Lines 16 and 17** to compute the one-off accuracies for both the training and validation/testing sets. These one-off accuracies are logged to our training history file (**Lines 20-23**). The outer `one_off_callback` function returns the `_callback` function to mxnet, allowing it to pass in the epoch number, model symbol, and relevant parameters after every epoch (**Line 26**).

The next step is to define the `_compute_one_off` function. As the name suggests, this method will be responsible for accepting an instantiated `model`, a data iterator (`dataIter`), and the `oneOff` dictionary mapping and then computing the one-off accuracies:

```
28 def _compute_one_off(model, dataIter, oneOff):
29     # initialize the total number of samples along with the
30     # number of correct (maximum of one off) classifications
31     total = 0
32     correct = 0
33
34     # loop over the predictions of batches
35     for (preds, _, batch) in model.iter_predict(dataIter):
36         # convert the batch of predictions and labels to NumPy
37         # arrays
38         predictions = preds[0].asnumpy().argmax(axis=1)
39         labels = batch.label[0].asnumpy().astype("int")
40
41         # loop over the predicted labels and ground-truth labels
42         # in the batch
43         for (pred, label) in zip(predictions, labels):
44             # if correct label is in the set of "one off"
45             # predictions, then update the correct counter
46             if label in oneOff[pred]:
47                 correct += 1
48
49             # increment the total number of samples
50             total += 1
51
52         # finish computing the one-off metric
53         return correct / float(total)
```

Line 31 initializes the `total` number of samples in the `dataIter`. **Line 32** then initializes the number of `correct` one-off predictions. To compute the number of correct one-off predictions, we must use the `iter_predict` method of the `model` to loop over samples in the `dataIter`, just as we did in Chapter 13 on identifying vehicle make and models (**Line 35**).

Line 38 grabs the output `predictions` from our model for the given batch, then finds the index of the label with the *largest probability*. On **Line 39** we extract the ground-truth `labels` so we can compare them to our `predictions`. We loop over each of the predicted and ground-truth labels on **Line 43**. If the ground-truth `label` exists in the set of one-off labels for the prediction (`pred`), then we increment the number of `correct` classifications.

We then increment the total number of samples examined in the `dataIter` on **Line 50**. Finally, **Line 53** returns the one-off metric as a ratio of the `correct` predictions to the `total` number of

data points.

If this callback seems a bit confusing at first glance, rest assured you're not the only developer who has struggled to create mxnet callbacks (myself included). If you find yourself struggling with this code, I suggest treating it as a "black box evaluation metric" and returning to it later once you see how it is used inside our `train.py` script.

I would also suggest that you read the mxnet documentation on *existing* callbacks (http://pyimg.co/d2u11) as well as reviewing the actual *implementations* of these callbacks (http://pyimg.co/fos3c). In particular, pay attention to the `module_checkpoint` function which provides an example of how to use the encapsulated `_callback` method.

14.5 Training Our Age and Gender Predictor

Now that we have our record dataset generated as well as our one-off evaluation metric implemented, let's move on to `train.py`, the script responsible for training *both* the age and gender CNNs. This script will be able to handle both the age and gender CNNs due to our `age_gender_config.py` file – whatever we set the `DATASET_TYPE` to, is what the CNN will bet trained on. Furthermore, the `DATASET_TYPE` also set all relevant output directories inside `age_gender_config`. This is yet another great example of the importance of using configuration files for deep learning projects.

Our `train.py` script will be near identical to all of our other projects in the *ImageNet Bundle*, so let's briefly review the file. First, we'll start with our imports:

```
1  # import the necessary packages
2  from config import age_gender_config as config
3  from pyimagesearch.nn.mxconv import MxAgeGenderNet
4  from pyimagesearch.utils import AgeGenderHelper
5  from pyimagesearch.mxcallbacks import one_off_callback
6  import mxnet as mx
7  import argparse
8  import logging
9  import pickle
10 import json
11 import os
```

The three most important imports to take note of here are our `MxAgeGenderNet` implementation, the `AgeGenderHelper`, and our `one_off_callback` (**Lines 3-5**). From there, we can parse our command line arguments as well as our our logging file:

```
13 # construct the argument parse and parse the arguments
14 ap = argparse.ArgumentParser()
15 ap.add_argument("-c", "--checkpoints", required=True,
16     help="path to output checkpoint directory")
17 ap.add_argument("-p", "--prefix", required=True,
18     help="name of model prefix")
19 ap.add_argument("-s", "--start-epoch", type=int, default=0,
20     help="epoch to restart training at")
21 args = vars(ap.parse_args())
22
23 # set the logging level and output file
24 logging.basicConfig(level=logging.DEBUG,
25     filename="training_{}.log".format(args["start_epoch"]),
26     filemode="w")
```

The `--checkpoints` switch controls the path to where we will store the serialized weights to MxAgeGenderNet after every epoch. The `--prefix` switch controls the name of the CNN. And finally, if we are restarting training from a previous epoch, we can specify exactly *which* epoch via `--start-epoch`.

We'll also need to set our `batchSize` based on the `BATCH_SIZE` and `NUM_DEVICES` in the `config` module:

```
28 # determine the batch and load the mean pixel values
29 batchSize = config.BATCH_SIZE * config.NUM_DEVICES
30 means = json.loads(open(config.DATASET_MEAN).read())
```

Line 30 handles loading our RGB means for either the age or gender dataset, respectively. Let's go ahead and create our training data iterator:

```
32 # construct the training image iterator
33 trainIter = mx.io.ImageRecordIter(
34     path_imgrec=config.TRAIN_MX_REC,
35     data_shape=(3, 227, 227),
36     batch_size=batchSize,
37     rand_crop=True,
38     rand_mirror=True,
39     rotate=7,
40     mean_r=means["R"],
41     mean_g=means["G"],
42     mean_b=means["B"],
43     preprocess_threads=config.NUM_DEVICES * 2)
```

Since our images are already pre-aligned in the Adience dataset, we don't want to apply *too much* rotation, so a `rotate` of ± 7 degrees seems appropriate here. We'll also randomly crop 227×227 regions from the input 256×256 image by specifying the `rand_crop` flag.

We'll also need a corresponding validation image iterator:

```
45 # construct the validation image iterator
46 valIter = mx.io.ImageRecordIter(
47     path_imgrec=config.VAL_MX_REC,
48     data_shape=(3, 227, 227),
49     batch_size=batchSize,
50     mean_r=means["R"],
51     mean_g=means["G"],
52     mean_b=means["B"])
```

Levi et al. used the SGD optimizer to train their network – we'll do the same with a base learning rate of $1e-4$, a momentum term of 0.9, and L2 weight decay of 0.0005:

```
54 # initialize the optimizer
55 opt = mx.optimizer.SGD(learning_rate=1e-3, momentum=0.9, wd=0.0005,
56     rescale_grad=1.0 / batchSize)
```

Let's also define the output path to our model checkpoints and initialize the model argument and auxiliary parameters, respectively:

```
# construct the checkpoints path, initialize the model argument and
# auxiliary parameters
checkpointsPath = os.path.sep.join([args["checkpoints"],
     args["prefix"]])
argParams = None
auxParams = None
```

Next, we can determine if we are starting training from scratch or loading from a specific epoch:

```
# if there is no specific model starting epoch supplied, then
# initialize the network
if args["start_epoch"] <= 0:
     # build the LeNet architecture
     print("[INFO] building network...")
     model = MxAgeGenderNet.build(config.NUM_CLASSES)

# otherwise, a specific checkpoint was supplied
else:
     # load the checkpoint from disk
     print("[INFO] loading epoch {}...".format(args["start_epoch"]))
     (model, argParams, auxParams) = mx.model.load_checkpoint(
          checkpointsPath, args["start_epoch"])
```

Regardless if we are training from scratch or restarting training from a specific epoch, the next step is to compile the model:

```
# compile the model
model = mx.model.FeedForward(
     ctx=[mx.gpu(2), mx.gpu(3)],
     symbol=model,
     initializer=mx.initializer.Xavier(),
     arg_params=argParams,
     aux_params=auxParams,
     optimizer=opt,
     num_epoch=110,
     begin_epoch=args["start_epoch"])
```

Again, I am using two GPUs to train our CNN; however, this experiment can *easily* be run with just a single GPU. We start initializing our callbacks and evaluation metrics below:

```
# initialize the callbacks and evaluation metrics
batchEndCBs = [mx.callback.Speedometer(batchSize, 10)]
epochEndCBs = [mx.callback.do_checkpoint(checkpointsPath)]
metrics = [mx.metric.Accuracy(), mx.metric.CrossEntropy()]
```

These callbacks will be applied regardless if we are training on the *age* dataset or the *gender* dataset. However, if we are specifically working with *age*, then we have extra work to do:

```
# check to see if the one-off accuracy callback should be used
if config.DATASET_MEAN == "age":
```

```
97      # load the label encoder, then build the one-off mappings for
98      # computing accuracy
99      le = pickle.loads(open(config.LABEL_ENCODER_PATH, "rb").read())
100     agh = AgeGenderHelper(config)
101     oneOff = agh.buildOneOffMappings(le)
102     epochEndCBs.append(one_off_callback(trainIter, valIter,
103         oneOff, mx.gpu(0)))
```

Line 99 loads our age `LabelEncoder` from disk while **Line 100** instantiates the `AgeGenderHelper`. Based off the `config` and `le`, we can generate the `oneOff` dictionary mappings.

Finally, **Lines 102 and 103** update the `epochEndCBs` list by appending the `one_off_calback` function. We supply the training data iterator (`trainIter`), validation data iterator (`valIter`), `oneOff` dictionary mappings, and device context (`mx.gpu(0)`) to the `one_off_callback`. These lines will enable us to both (1) compute the one-off evaluation metric for both the training and validation set, and (2) log the results to our training history file as well.

You'll also notice that I am using a separate GPU for my context. When working with mxnet, I found that for optimal performance, a separate GPU/CPU/device should be used for evaluation than the one(s) used for training. I'm not entirely sure why it's faster, but I do notice that evaluation slows down if you try to use the same context for both training and evaluation using the `one_off_callback` – this discrepancy in evaluation speed is something to keep in mind when developing your own custom evaluation metrics.

Our last step is to simply train the network:

```
105 # train the network
106 print("[INFO] training network...")
107 model.fit(
108     X=trainIter,
109     eval_data=valIter,
110     eval_metric=metrics,
111     batch_end_callback=batchEndCBs,
112     epoch_end_callback=epochEndCBs)
```

In our next section, we'll create `test_accuracy.py` so we can evaluate our MxAgeGenderNet weights on the respective testing sets. Then in Section 14.7, we'll apply both `train.py` and `test_accuracy.py` to run experiments on age and gender prediction.

14.6 Evaluating Age and Gender Prediction

Just as `train.py` can be used to *train* a network on either the age or gender datasets, our `test_accuracy.py` script will be able to *evaluate accuracy* on both the age and gender datasets – to switch between datasets, we simply need to update `DATASET_TYPE` in `age_gender_config`. Let's review `test_accuracy.py` now:

```
1 # since 'AgeGenderHelper' also imports OpenCV, we need to place it
2 # above the mxnet import to avoid a segmentation fault
3 from pyimagesearch.utils import AgeGenderHelper
4
5 # import the necessary packages
6 from config import age_gender_config as config
7 from pyimagesearch.mxcallbacks.mxmetrics import _compute_one_off
8 import mxnet as mx
```

```
9  import argparse
10 import pickle
11 import json
12 import os
```

Lines 3-12 take care of our Python imports. Since the `AgeGenderHelper` class imports the `cv2` library, I have placed it at the top of the file. As I mentioned earlier in this book, on two out of three machines I configured mxnet + OpenCV on, importing `cv2` before `mxnet` caused a segmentation fault. Placing any `cv2` imports prior to `mxnet` imports resolved the issue. It is unlikely that you will run into this problem, but if you do, simply follow the suggestions recommended above to resolve the problem.

Let's now parse our command line arguments:

```
14 # construct the argument parse and parse the arguments
15 ap = argparse.ArgumentParser()
16 ap.add_argument("-c", "--checkpoints", required=True,
17      help="path to output checkpoint directory")
18 ap.add_argument("-p", "--prefix", required=True,
19      help="name of model prefix")
20 ap.add_argument("-e", "--epoch", type=int, required=True,
21      help="epoch # to load")
22 args = vars(ap.parse_args())
23
24 # load the RGB means for the training set
25 means = json.loads(open(config.DATASET_MEAN).read())
```

You've seen all these command line arguments in previous chapters in the *ImageNet Bundle*. The `--checkpoints` switch controls the base directory to where our serialized MxAgeGenderNet weights live. The `--prefix` is the name of our the network. And finally, `--epoch` controls the integer value of the epoch we want to load from disk. When combined together, we can load a *specific* weight file from disk. **Line 25** then loads our RGB means so we can perform mean normalization.

Accessing the testing set requires constructing an `ImageRecordIter`:

```
27 # construct the testing image iterator
28 testIter = mx.io.ImageRecordIter(
29      path_imgrec=config.TEST_MX_REC,
30      data_shape=(3, 227, 227),
31      batch_size=config.BATCH_SIZE,
32      mean_r=means["R"],
33      mean_g=means["G"],
34      mean_b=means["B"])
```

From there, we can load the specific `--epoch` from disk:

```
36 # load the checkpoint from disk
37 print("[INFO] loading model...")
38 checkpointsPath = os.path.sep.join([args["checkpoints"],
39      args["prefix"]])
40 model = mx.model.FeedForward.load(checkpointsPath,
```

```
41        args["epoch"])
42
43  # compile the model
44  model = mx.model.FeedForward(
45        ctx=[mx.gpu(0)],
46        symbol=model.symbol,
47        arg_params=model.arg_params,
48        aux_params=model.aux_params)
```

Regardless if we are evaluating our network on the age or gender dataset, we need to compute the rank-1 accuracy, which is handled by the following code block:

```
50  # make predictions on the testing data
51  print("[INFO] predicting on '{}' test data...".format(
52        config.DATASET_TYPE))
53  metrics = [mx.metric.Accuracy()]
54  acc = model.score(testIter, eval_metric=metrics)
55
56  # display the rank-1 accuracy
57  print("[INFO] rank-1: {:.2f}%".format(acc[0] * 100))
```

However, if we are working with the age dataset, we also need to compute the one-off accuracy as well:

```
59  # check to see if the one-off accuracy callback should be used
60  if config.DATASET_TYPE == "age":
61        # re-compile the model so that we can compute our custom one-off
62        # evaluation metric
63        arg = model.arg_params
64        aux = model.aux_params
65        model = mx.mod.Module(symbol=model.symbol, context=[mx.gpu(1)])
66        model.bind(data_shapes=testIter.provide_data,
67              label_shapes=testIter.provide_label)
68        model.set_params(arg, aux)
69
70        # load the label encoder, then build the one-off mappings for
71        # computing accuracy
72        le = pickle.loads(open(config.LABEL_ENCODER_PATH, "rb").read())
73        agh = AgeGenderHelper(config)
74        oneOff = agh.buildOneOffMappings(le)
75
76        # compute and display the one-off evaluation metric
77        acc = _compute_one_off(model, testIter, oneOff)
78        print("[INFO] one-off: {:.2f}%".format(acc * 100))
```

On **Line 60** we make a check to ensure we are evaluating our network on the `age` dataset. Provided that we are, we unpack the `model` parameters on **Lines 63 and 64**, followed by re-initializing the `model` as a `Module` object that can be compatible with our one-off accuracy metric (**Lines 65-68**).

Lines 72-74 build our `oneOff` dictionary mappings. A call to `_compute_one_off` on **Line 77** computes the one-off accuracy for our testing set. We use the `_compute_one_off` function here rather than the `one_off_callback` function because we need to evaluate accuracy on a

single data iterator (keep in mind that the `one_off_callback` function requires *two* data iterators, one presumed to be the training iterator and the other a validation/testing iterator). Finally, **Line 78** prints the one-off accuracy to our terminal.

In our next section, we'll see how both `train.py` and `test_accuracy.py` are used together to train and evaluate two CNNs for age and gender prediction accuracy, respectively.

14.7 Age and Gender Prediction Results

In previous chapters in both the *Practitioner Bundle* and *ImageNet Bundle*, I provided a series of experiments to demonstrate how I obtained an optimally performing model. However, due to the length of this chapter, I am going to present the *best results only* for both the age CNN and gender CNN.

14.7.1 Age Results

My best experiment on predicting age using MxAgeGenderNet was obtained using the Adam optimizer with an initial learning rate of $1e-4$ (which is smaller than the default $1e-3$). An L2 weight decay of 0.0005 was also applied. Using a small initial learning rate enabled me to train the network for *longer* without overfitting creeping in. I started training using the following command:

```
$ python train.py --checkpoints checkpoints/age --prefix agenet
```

The gap between training and validation remains small until approximately epoch 90 when divergence starts. I allowed training to continue to epoch 120 (Figure 14.3, *top-left*), where I stopped training, lowered the learning rate to $1e-5$, and restarted training:

```
$ python train.py --checkpoints checkpoints/age --prefix agenet \
    --start-epoch 120
```

I allowed training to continue for 20 epochs (Figure 14.3, *top-right*). At this point, we can start to see both the validation loss and accuracy start to level out a bit. I didn't want to train for too much longer as overfitting to the training data was a concern of mine. I decided to lower the learning rate to $1e-6$ and then resume training for 10 more epochs:

```
$ python train.py --checkpoints checkpoints/age --prefix agenet \
    --start-epoch 140
```

After epoch 150, I killed the training process and examined my validation set, noting the validation accuracy to be 71.65% rank-1 and 94.14% one-off (Figure 14.3, *bottom*). Happy with these results, I then evaluated on the testing set:

```
$ python test_accuracy.py --checkpoints checkpoints/age --prefix agenet \
    --epoch 150
[INFO] loading model...
[INFO] predicting on 'age' test data...
[INFO] rank-1: 71.15%
[INFO] one-off: 88.28%
```

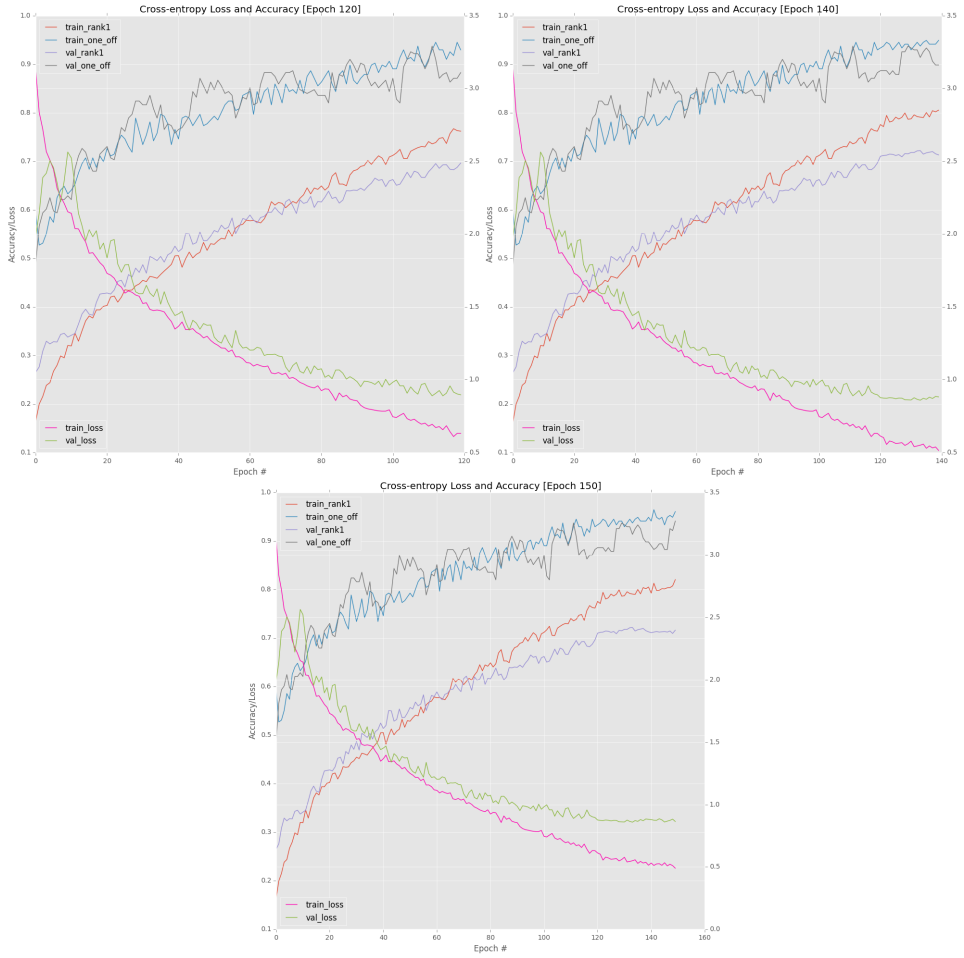


Figure 14.3: **Top-left:** Using an initial learning rate of $\alpha 1e-4$ allows us to slowly but steadily train up until epoch 120. **Top-right:** At this point training and validation metrics start to deviate from each other so we adjust α to $1e-5$ and train for another 10 epochs. **Bottom:** Using $\alpha = 1e-6$ stagnates training.

Here you can see that this approach yielded **71.15**% rank-1 and **88.28**% one-off accuracy on the testing set. Compared to the original method by Levi et al., this result is a *substantial improvement*. Their best method (using 10-crop oversampling) achieved only achieved 50.7% rank-1 accuracy and 84.70% one-off accuracy. The rank-1 results alone in this experiment are an improvement by a rate of **20.45**%!

14.7.2 Gender Results

After applying `train.py` to the age dataset, I went back to `age_gender_config` and set `DATASET_TYPE` to `gender` to indicate that I wanted to train the MxAgeGenderNet architecture on the gender dataset. My best results came from training using the SGD optimizer with a base learning rate of $1e-2$. I also applied a momentum term of 0.9 and L2 weight decay of 0.0005. I started training using the following command:

```
$ python train.py --checkpoints checkpoints/gender --prefix gendernet
```

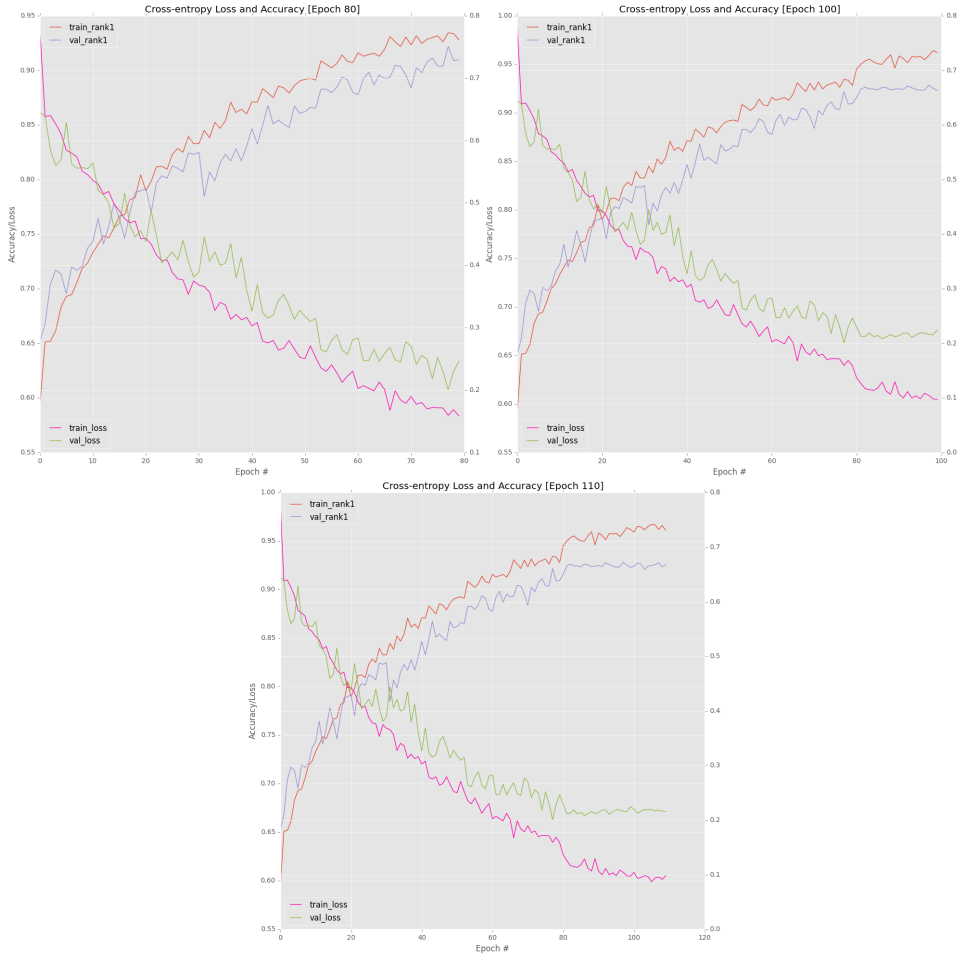


Figure 14.4: **Top-left:** The first 80 epochs training our network on the gender dataset. **Top-right:** Lowering α from $1e-2$ to $1e-3$ for 20 epochs. **Bottom:** The final 10 epochs at $\alpha = 1e-4$.

As the image below shows, training is slow and steady (Figure 14.4, *top-left*). At epoch 80, I stopped training, lowered the learning rate to $1e-3$ and continued training for another 20 epochs:

```
$ python train.py --checkpoints checkpoints/gender --prefix gendernet \
    --start-epoch 80
```

The output plot of epoch 100 can be seen in Figure 14.4 *(top-right)*. My biggest concern here was overfitting. The validation loss had leveled out entirely with the training loss continuing to decrease. I didn't want to train for much longer as overfitting seemed plausible, so I lowered the learning rate from $1e-3$ to $1e-4$ and allowed the network to train for 10 more epochs:

```
$ python train.py --checkpoints checkpoints/gender --prefix gendernet \
    --start-epoch 100
```

The final plot of all 120 epochs can be seen in Figure 14.4 *(bottom)*. Looking at the loss plot, we can see the stagnation in validation, while training continues to decrease. However, when

examining the accuracy plot, we note that the gap between training and validation is relatively stable for the majority of the training process. The validation accuracy after epoch 120 was 92.57%.

I then used epoch 120 to evaluate the test set:

```
$ python test_accuracy.py --checkpoints checkpoints/gender \
    --prefix gendernet --epoch 110
[INFO] loading model...
[INFO] predicting on 'gender' test data...
[INFO] rank-1: 90.29%
```

As you can see, I achieved **90.29%** accuracy on the testing set. Looking at the results of Levi et al., note that their approach (using the 10-crop oversampling technique) yielded 86.8% accuracy – my method is a full 3.5% higher.

Now that both of our age and gender Convolutional Neural Networks are trained and evaluated, let's move on to how we can *prepare* and *pre-process* images for classification using these networks.

14.8 Visualizing Results

Now that we have trained two separate CNNs – one for age prediction and the other for gender identification – we can move on to the topic of *deployment*. During the deployment phase, we'll need to be able to load both of our pre-trained CNNs, pre-process our input images, and then pass them through the two networks to obtain our output classifications.

Instead of using the existing `age_gender_config` configuration, let's instead open up a new file, name it `age_gender_deploy.py`, and insert the following configurations:

```
1  # import the necessary packages
2  from age_gender_config import OUTPUT_BASE
3  from os import path
4
5  # define the path to the dlib facial landmark predictor
6  DLIB_LANDMARK_PATH = "shape_predictor_68_face_landmarks.dat"
```

In order to obtain higher accuracy when predicting age and gender, it's often helpful to *crop* and *align* a face from a given image. Thus far, all of our input images have been pre-cropped and pre-aligned for us; however, this assumption will not hold in the real-world. To boost accuracy further, we need to apply *face alignment*, a topic we'll discuss in Section 14.8.2 below. The `DLIB_LANDMARK_PATH` variables provideacc the path to a pre-trained facial landmark predictor that will enable us to align faces in input images.

Next, let's define the age CNN configurations:

```
8  # define the path to the age network + supporting files
9  AGE_NETWORK_PATH = "checkpoints/age"
10 AGE_PREFIX = "agenet"
11 AGE_EPOCH = 150
12 AGE_LABEL_ENCODER = path.sep.join([OUTPUT_BASE, "age_le.cpickle"])
13 AGE_MEANS = path.sep.join([OUTPUT_BASE, "age_adience_mean.json"])
```

Here we start by defining the `AGE_NETWORK_PATH`, which is the path to the age weight checkpoints. The `AGE_PREFIX` controls the name of the network, in this case `agenet`. We'll be loading

AGE_EPOCH number 150 from disk. To convert from raw integer labels to human-readable labels, we'll need path to our AGE_LABEL_ENCODER. Finally, to perform mean normalization we need the AGE_MEANS path.

We define identical variables for the gender network below:

```
15 # define the path to the gender network + supporting files
16 GENDER_NETWORK_PATH = "checkpoints/gender"
17 GENDER_PREFIX = "gendernet"
18 GENDER_EPOCH = 110
19 GENDER_LABEL_ENCODER = path.sep.join([OUTPUT_BASE,
20     "gender_le.cpickle"])
21 GENDER_MEANS = path.sep.join([OUTPUT_BASE,
22     "gender_adience_mean.json"])
```

In our next section we'll learn how to apply our two networks to classify and visualize the results of age and gender predictions from images *inside* the Adience dataset. But what happens if you want to apply these types of predictions to images *outside* Adience? To do so, we first need to understand the process of *facial alignment*. From there, we'll use these techniques (and our pre-trained networks) to apply age and gender predictions to our own custom images.

14.8.1 Visualizing Results from Inside Adience

Let's go ahead and review vis_classification.py, the script responsible for visualizing predictions *inside* the Adience dataset. Open up the vis_classification.py file and we'll get to work:

```
1  # import OpenCV before mxnet to avoid a segmentation fault
2  import cv2
3
4  # import the necessary packages
5  from config import age_gender_config as config
6  from config import age_gender_deploy as deploy
7  from pyimagesearch.preprocessing import ImageToArrayPreprocessor
8  from pyimagesearch.preprocessing import SimplePreprocessor
9  from pyimagesearch.preprocessing import MeanPreprocessor
10 from pyimagesearch.utils import AgeGenderHelper
11 import numpy as np
12 import mxnet as mx
13 import argparse
14 import pickle
15 import imutils
16 import json
17 import os
```

We start off by importing our necessary Python packages on **Lines 2-17** – notice how I am once again placing the cv2 import above mxnet, which is because of the seg-fault issue I mentioned above. On your machine, this action may not be required. From there, we import both our standard config and deploy configuration files.

Lines 7-9 import our image pre-processors so we can properly pre-process our images before classifying them. The AgeGenderHelper will be used to visualize the probability distributions for each class label.

Let's move on to the command line arguments:

```
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-s", "--sample-size", type=int, default=10,
	help="epoch # to load")
args = vars(ap.parse_args())
```

We only need a single switch here, `--sample-size`, which is an integer representing the number of images we want to sample from the Adience testing set. We can now load our label encoders and mean files for both the age and gender datasets:

```
# load the label encoders and mean files
print("[INFO] loading label encoders and mean files...")
ageLE = pickle.loads(open(deploy.AGE_LABEL_ENCODER, "rb").read())
genderLE = pickle.loads(open(deploy.GENDER_LABEL_ENCODER, "rb").read())
ageMeans = json.loads(open(deploy.AGE_MEANS).read())
genderMeans = json.loads(open(deploy.GENDER_MEANS).read())
```

As well as load the serialized networks from disk:

```
# load the models from disk
print("[INFO] loading models...")
agePath = os.path.sep.join([deploy.AGE_NETWORK_PATH,
	deploy.AGE_PREFIX])
genderPath = os.path.sep.join([deploy.GENDER_NETWORK_PATH,
	deploy.GENDER_PREFIX])
ageModel = mx.model.FeedForward.load(agePath, deploy.AGE_EPOCH)
genderModel = mx.model.FeedForward.load(genderPath,
	deploy.GENDER_EPOCH)
```

Once the models are loaded we need to compile them:

```
# now that the networks are loaded, we need to compile them
print("[INFO] compiling models...")
ageModel = mx.model.FeedForward(ctx=[mx.gpu(0)],
	symbol=ageModel.symbol, arg_params=ageModel.arg_params,
	aux_params=ageModel.aux_params)
genderModel = mx.model.FeedForward(ctx=[mx.gpu(0)],
	symbol=genderModel.symbol, arg_params=genderModel.arg_params,
	aux_params=genderModel.aux_params)
```

Before passing an image through a given network, the image must first be pre-processed. Let's initialize those pre-preprocessors now:

```
# initialize the image pre-processors
sp = SimplePreprocessor(width=227, height=227, inter=cv2.INTER_CUBIC)
ageMP = MeanPreprocessor(ageMeans["R"], ageMeans["G"],
	ageMeans["B"])
genderMP = MeanPreprocessor(genderMeans["R"], genderMeans["G"],
	genderMeans["B"])
iap = ImageToArrayPreprocessor()
```

Notice how we are instantiating *two* `MeanPreprocessor` objects: one for the *age* and one for the *gender*. We use two mean processors here because both age and gender have *two separate training sets* and therefore have different RGB mean values.

The following code block samples `--sample-size` rows from the testing `.lst` file:

```
59 # load a sample of testing images
60 rows = open(config.TEST_MX_LIST).read().strip().split("\n")
61 rows = np.random.choice(rows, size=args["sample_size"])
```

Let's loop over each of these `rows` individually:

```
63 # loop over the rows
64 for row in rows:
65     # unpack the row
66     (_, gtLabel, imagePath) = row.strip().split("\t")
67     image = cv2.imread(imagePath)
68
69     # pre-process the image, one for the age model and another for
70     # the gender model
71     ageImage = iap.preprocess(ageMP.preprocess(
72         sp.preprocess(image)))
73     genderImage = iap.preprocess(genderMP.preprocess(
74         sp.preprocess(image)))
75     ageImage = np.expand_dims(ageImage, axis=0)
76     genderImage = np.expand_dims(genderImage, axis=0)
```

For each `row`, we unpack it into the ground-truth label (`gtLabel`) and corresponding `imagePath` (**Line 66**). The input `image` is loaded from disk on **Line 67**. Pre-processing the image is handled on **Lines 71-76**, creating two output images.

The first output image is `ageImage`, the result of the `image` being processed by the age pre-processors. Similarly, we have the `genderImage`, the result of being passed through the gender pre-processors. Now that our images have been pre-processed, we can pass them through their respective networks for classification:

```
78     # pass the ROIs through their respective models
79     agePreds = ageModel.predict(ageImage)[0]
80     genderPreds = genderModel.predict(genderImage)[0]
81
82     # sort the predictions according to their probability
83     ageIdxs = np.argsort(agePreds)[::-1]
84     genderIdxs = np.argsort(genderPreds)[::-1]
```

A call to the `.predict` methods on **Lines 79 and 80** give us our probabilities for each class label. We then sort these labels in descending order with the largest probability at the front of the lists (**Lines 83 and 84**).

To visualize the probability distribution of each class label, we'll use our `visualizeAge` and `visualizeGender` methods:

```
86     # visualize the age and gender predictions
87     ageCanvas = AgeGenderHelper.visualizeAge(agePreds, ageLE)
```

Figure 14.5: Various example classifications from within the Adience dataset used our age and gender networks. In each of the cases we were able to correctly predict both the *age* and *gender* of the subject.

```
88      genderCanvas = AgeGenderHelper.visualizeGender(genderPreds,
89          genderLE)
90      image = imutils.resize(image, width=400)
```

For the sake of brevity, these function explanations were not included in this chapter. These two functions are simply extensions to the emotion/facial expression visualizations in Chapter 11. For those interested, I have provided an explanation of these two functions in the companion website to this book. Otherwise, I will leave it as an exercise to the reader to investigate these functions as they are *totally unrelated to deep learning* and simply *OpenCV functions* used to draw a nicely formatted probability distribution.

Our final code block draws both the top *age* and *gender* prediction on the output `image`, followed by displaying the results to our screen:

```
92      # draw the actual prediction on the image
93      gtLabel = ageLE.inverse_transform(int(gtLabel))
94      text = "Actual: {}-{}".format(*gtLabel.split("_"))
95      cv2.putText(image, text, (10, 30), cv2.FONT_HERSHEY_SIMPLEX,
96          0.7, (0, 0, 255), 3)
97
98      # show the output image
99      cv2.imshow("Image", image)
100     cv2.imshow("Age Probabilities", ageCanvas)
101     cv2.imshow("Gender Probabilities", genderCanvas)
102     cv2.waitKey(0)
```

To see our `vis_classification.py` script in action, open up a terminal and execute the following command:

```
$ python vis_classification.py
```

A sample of the output results can be seen in Figure 14.5. Notice how we are able to correctly predict both the *age* and *gender* of the person in the photo.

14.8.2 Understanding Face Alignment

In order to understand face alignment, we first need to review the fundamentals of *facial landmarks*. Facial landmarks are used to localize and represent the salient regions of the face, such as:

- Eyes
- Eyebrows
- Nose
- Mouth
- Jawline

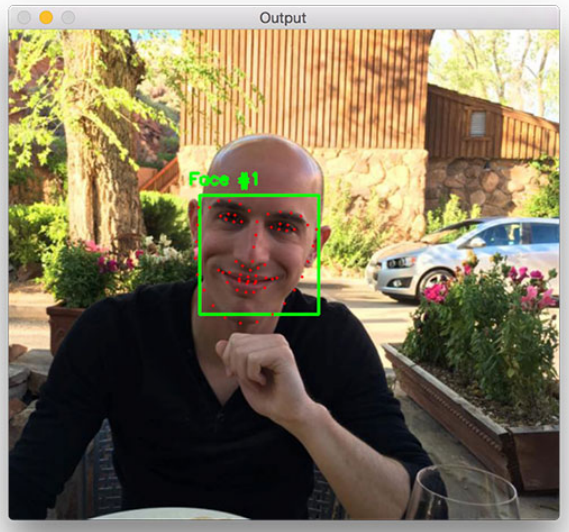


Figure 14.6: An example of detecting facial landmarks on a face. Notice how the eyes, eyebrows, nose, mouth, and jawline are localized.

An example of facial landmarks detected on an image can be seen in Figure 14.6. Detecting facial landmarks is a *subset* of the shape prediction problem. Given an input image (normally a ROI that specifies the object of interest), a shape predictor attempts to localize key points of interest along the shape. In the context of facial landmarks, our goal is to detect important facial structures on the face using shape prediction models. Detecting facial landmarks is, therefore, a two step process:

- **Step #1:** Localize the face in the image.
- **Step #2:** Detect the key facial structures on the face ROI.

To accomplish both of these steps, we will be using the dlib library [40]. For Step #1, dlib uses a pre-trained HOG + Linear SVM [41, 42] detector – this detector can locate the bounding box (x, y)-coordinates of a face in an image. Then, for Step #2, using the method proposed by Kazemi and Sullivan in their 2014 paper, *One Millisecond Face Alignment with an Ensemble of Regression*

Trees, the key facial structures can be localized. The end result is a facial landmark detector that can be used to detect facial landmarks in real-time with high-quality results.

Given the facial landmarks, we can then apply *facial alignment*, the process of:

1. Identifying the geometric structure of faces in digital images.
2. Attempting to obtain a canonical alignment of the face based on translation, scale, and rotation.

There are many forms of face alignment. Some methods try to impose a (pre-defined) 3D model and then apply a transformation to the input images such that the facial landmarks on the input face match the landmarks on the 3D Model. Other, more simplistic methods, rely on the facial landmarks themselves (in particular, *the eye regions*) to obtain a normalized rotation, translation, and scale representation of the face.

Thus, face alignment can be seen as yet another form of *data normalization*. Just as we mean normalize images prior to passing them through a CNN, we can align faces to obtain better accuracy when (1) training our CNNs and (2) evaluating them. A full review of the facial alignment process is outside the scope of this book as it focuses heavily on computer vision and image processing techniques *outside* the deep learning space; however, I encourage readers interested in both facial landmarks and face alignments to read my tutorials on these methods:

- **Facial landmarks:** http://pyimg.co/xkgwd
- **Face alignment:** http://pyimg.co/tnbzf

In order to apply facial alignment to our input images, we'll be using the `FaceAligner` class implemented in my `imutils` library (http://pyimg.co/7c29j). I briefly describe the face alignment algorithm below.

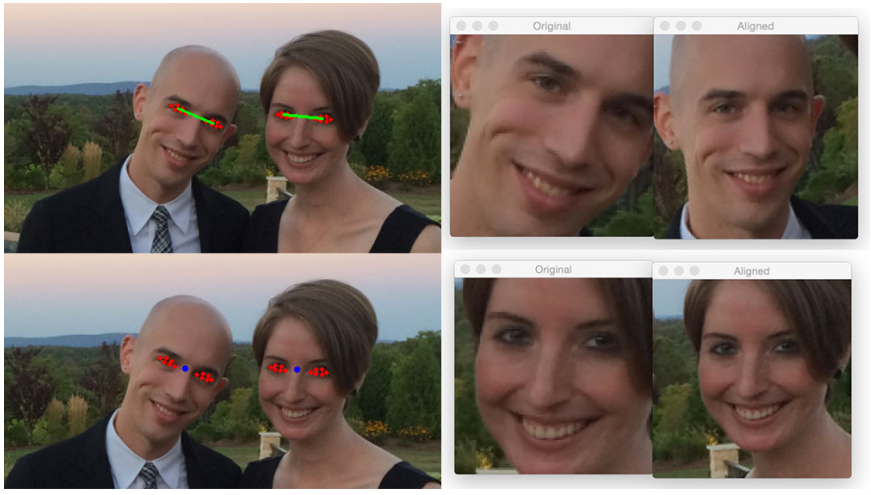


Figure 14.7: **Top:** Localizing the eyes via facial landmarks (red) is followed by computing the angle between the eyes (green). **Bottom:** In order to align the face we also need to compute the center (x, y) midpoint between the eyes. **Right:** Results of applying facial alignment to the two faces. Notice how both faces are scaled approximately equally and rotated such at the eyes lie along a horizontal line.

First, the eyes are localized using facial landmarks (Figure 14.7, *top*). Given the (x, y)-coordinates of the eyes, we can then compute the angle between them. From there we also need the midpoint between the eyes (*bottom*). An affine transformation is then applied to warp the images into a new output coordinate space, such that the face is:

1. Centered in the image.

2. Rotated such that the eyes lie on a horizontal line (i.e., the face is rotated such that the eyes lie along the same y-coordinates).
3. Scaled such that the size of all faces across a given dataset are approximately equal.

The results of applying facial alignment to the two faces can be seen in Figure 14.7 (*right*). Notice how in each example the face is centered, scaled approximately equally, and rotated such that the eyes lie along a horizontal line. Again, a full review of the facial landmarks are outside the scope of this book, so please refer to the PyImageSearch blog links above for more information on both of these methods. In the remainder of this chapter, we'll treat facial alignment as a blackbox algorithm. Those interested in learning more about these techniques should refer to the links.

14.8.3 Applying Age and Gender Prediction to Your Own Images

Being able to apply age and gender prediction to the Adience dataset is all well and good – but if what if we wanted to apply our CNNs to images *outside* of Adience? What do we do then? And what are the necessary steps to take? To find out, open the `test_prediction.py`, and start by importing the following libraries:

```
1  # import OpenCV before mxnet to avoid a segmentation fault
2  import cv2
3
4  # import the necessary packages
5  from config import age_gender_deploy as deploy
6  from pyimagesearch.preprocessing import ImageToArrayPreprocessor
7  from pyimagesearch.preprocessing import SimplePreprocessor
8  from pyimagesearch.preprocessing import MeanPreprocessor
9  from pyimagesearch.preprocessing import CropPreprocessor
10 from pyimagesearch.utils import AgeGenderHelper
11 from imutils.face_utils import FaceAligner
12 from imutils import face_utils
13 from imutils import paths
14 import numpy as np
15 import mxnet as mx
16 import argparse
17 import pickle
18 import imutils
19 import json
20 import dlib
21 import os
```

As you can see, there are quite a number of Python packages that we need to import. To start, we'll need our `deploy` configuration so we can load both of our age and gender networks. We'll be using a number of image preprocessors. Take note of **Line 9** where we import the *CropPreprocessor* – to boost classification accuracy, we'll be performing the 10-crop method from Chapter 10 of the *Practitioner Bundle*.

We then have the `FaceAligner` class imported on **Line 11**. The face alignment method was briefly discussed in Section 14.8.2 above (please see http://pyimg.co/tnbzf for more information), but for the time being, simply treat this class as a blackbox face alignment tool.

We then have our command line arguments:

```
23 # construct the argument parse and parse the arguments
24 ap = argparse.ArgumentParser()
25 ap.add_argument("-i", "--image", required=True,
```

```
26      help="path to input image (or directory)")
27  args = vars(ap.parse_args())
```

Only a single argument is needed here, `--image`, which can either be:

1. A path to a *single* image.
2. A path to a *directory* of images.

In the case that `--image` is a directory, we'll loop over each of the images individually and apply age and gender classification to each of them. Our next few code blocks will look similar to our previous `vis_classification.py` script. We'll start by loading our label encoders and mean files:

```
29  # load the label encoders and mean files
30  print("[INFO] loading label encoders and mean files...")
31  ageLE = pickle.loads(open(deploy.AGE_LABEL_ENCODER, "rb").read())
32  genderLE = pickle.loads(open(deploy.GENDER_LABEL_ENCODER, "rb").read())
33  ageMeans = json.loads(open(deploy.AGE_MEANS).read())
34  genderMeans = json.loads(open(deploy.GENDER_MEANS).read())
```

Followed by loading the serialized networks themselves:

```
36  # load the models from disk
37  print("[INFO] loading models...")
38  agePath = os.path.sep.join([deploy.AGE_NETWORK_PATH,
39      deploy.AGE_PREFIX])
40  genderPath = os.path.sep.join([deploy.GENDER_NETWORK_PATH,
41      deploy.GENDER_PREFIX])
42  ageModel = mx.model.FeedForward.load(agePath, deploy.AGE_EPOCH)
43  genderModel = mx.model.FeedForward.load(genderPath,
44      deploy.GENDER_EPOCH)
```

Once the models have been loaded, they also need to be compiled:

```
46  # now that the networks are loaded, we need to compile them
47  print("[INFO] compiling models...")
48  ageModel = mx.model.FeedForward(ctx=[mx.gpu(0)],
49      symbol=ageModel.symbol, arg_params=ageModel.arg_params,
50      aux_params=ageModel.aux_params)
51  genderModel = mx.model.FeedForward(ctx=[mx.gpu(0)],
52      symbol=genderModel.symbol, arg_params=genderModel.arg_params,
53      aux_params=genderModel.aux_params)
```

Of course, we'll need to pre-process our images before passing them through the networks:

```
55  # initialize the image pre-processors
56  sp = SimplePreprocessor(width=256, height=256,
57      inter=cv2.INTER_CUBIC)
58  cp = CropPreprocessor(width=227, height=227, horiz=True)
59  ageMP = MeanPreprocessor(ageMeans["R"], ageMeans["G"],
60      ageMeans["B"])
61  genderMP = MeanPreprocessor(genderMeans["R"], genderMeans["G"],
62      genderMeans["B"])
63  iap = ImageToArrayPreprocessor(dataFormat="channels_first")
```

Notice how we have initialized our `SimplePreprocessor` to resize an image to 256×256 pixels (**Line 56 and 57**). Once an image has been resized, we'll apply the 10-crop method, extracting 227×227 regions from the input image using the `CropPreprocessor` (**Line 58**). Special care is taken on **Line 63** to instantiate the `ImageToArrayPreprocessor` with the "channels first" `dataFormat`. We use "channels first" ordering due to the fact that mxnet represents images with the channels *before* the spatial dimensions.

The next code block handles initializing dlib's (HOG-based [41, 42]) face detector, loading the facial landmark predictor from disk, and instantiating our `FaceAligner`:

```
65 # initialize dlib's face detector (HOG-based), then create the
66 # the facial landmark predictor and face aligner
67 detector = dlib.get_frontal_face_detector()
68 predictor = dlib.shape_predictor(deploy.DLIB_LANDMARK_PATH)
69 fa = FaceAligner(predictor)
```

Next, we should determine if we are loading a single image from disk, or if we should list the contents of the directory and grab the paths to all input images:

```
71 # initialize the list of image paths as just a single image
72 imagePaths = [args["image"]]
73
74 # if the input path is actually a directory, then list all image
75 # paths in the directory
76 if os.path.isdir(args["image"]):
77     imagePaths = sorted(list(paths.list_files(args["image"])))
```

Given our `imagePaths`, let's loop over them individually:

```
79 # loop over the image paths
80 for imagePath in imagePaths:
81     # load the image from disk, resize it, and convert it to
82     # grayscale
83     print("[INFO] processing {}".format(imagePath))
84     image = cv2.imread(imagePath)
85     image = imutils.resize(image, width=800)
86     gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
87
88     # detect faces in the grayscale image
89     rects = detector(gray, 1)
```

We start by loading our image from disk (**Line 84**), resizing it to have a fixed width of 800 pixels (**Line 85**), and converting it to grayscale (**Line 86**). **Line 89** then uses dlib's face detector to detect the locations of faces in the image.

Let's loop over each of the detected faces:

```
91     # loop over the face detections
92     for rect in rects:
93         # determine the facial landmarks for the face region, then
94         # align the face
95         shape = predictor(gray, rect)
```

```
96          face = fa.align(image, gray, rect)
97
98          # resize the face to a fixed size, then extract 10-crop
99          # patches from it
100         face = sp.preprocess(face)
101         patches = cp.preprocess(face)
102
103         # allocate memory for the age and gender patches
104         agePatches = np.zeros((patches.shape[0], 3, 227, 227),
105             dtype="float")
106         genderPatches = np.zeros((patches.shape[0], 3, 227, 227),
107             dtype="float")
```

For each of the bounding box face location, `rect`, we apply the facial landmark predictor, extract the face, and align it (**Lines 95 and 96**). We then pre-preprocess the face by applying `SimplePreprocessor` to resize it to 256×256 pixels, followed by the `CropPreprocessor` to extract the 10 `patches` of the `face`.

Lines 104-107 initialize empty NumPy arrays to store each of the ten crops for both the age predictions and gender predictions, respectively. The reason we *explicitly* define these two arrays is because each ROI in `patches` needs to be pre-processed by both the age pre-processors and gender pre-processors. We can apply these pre-processors now:

```
109         # loop over the patches
110         for j in np.arange(0, patches.shape[0]):
111             # perform mean subtraction on the patch
112             agePatch = ageMP.preprocess(patches[j])
113             genderPatch = genderMP.preprocess(patches[j])
114             agePatch = iap.preprocess(agePatch)
115             genderPatch = iap.preprocess(genderPatch)
116
117             # update the respective patches lists
118             agePatches[j] = agePatch
119             genderPatches[j] = genderPatch
```

On **Line 110** we start looping over each of the ten patches. For each one, we apply the age mean subtraction pre-processor and the gender mean subtraction pre-processor (**Lines 112 and 113**). Both `agePatch` and `genderPatch` are then converted to mxnet-compatible arrays via channel ordering on **Lines 114 and 115**. After both have been through the pre-processing pipeline, they can be added to the `agePatches` and `genderPatches` arrays, respectively (**Lines 118 and 119**).

We are now finally ready to make predictions on the `agePatches` and `genderPatches`:

```
121         # make predictions on age and gender based on the extracted
122         # patches
123         agePreds = ageModel.predict(agePatches)
124         genderPreds = genderModel.predict(genderPatches)
125
126         # compute the average for each class label based on the
127         # predictions for the patches
128         agePreds = agePreds.mean(axis=0)
129         genderPreds = genderPreds.mean(axis=0)
```

Lines 123 and 124 pass the age and gender ROIs through their respective networks, yielding the class probabilities for each patch. We average the class probabilities together across every patch to obtain our final predictions on **Lines 128 and 129**.

Our final code block handles displaying the class label distribution for both age and gender, as well as drawing a bounding box surrounding the face we are predicting the age and gender for:

```
131             # visualize the age and gender predictions
132             ageCanvas = AgeGenderHelper.visualizeAge(agePreds, ageLE)
133             genderCanvas = AgeGenderHelper.visualizeGender(genderPreds,
134                  genderLE)
135
136             # draw the bounding box around the face
137             clone = image.copy()
138             (x, y, w, h) = face_utils.rect_to_bb(rect)
139             cv2.rectangle(clone, (x, y), (x + w, y + h), (0, 255, 0), 2)
140
141             # show the output image
142             cv2.imshow("Input", clone)
143             cv2.imshow("Face", face)
144             cv2.imshow("Age Probabilities", ageCanvas)
145             cv2.imshow("Gender Probabilities", genderCanvas)
146             cv2.waitKey(0)
```

The final output images are displayed to our screen on **Lines 142-146**.

To apply our `test_prediction.py` script on images outside of the Adience dataset, simply execute the following command:

```
$ python test_prediction.py --image examples
```

A set of example results can be found in Figure 14.8. Notice how we were able to correctly classify the subject in the photo based on their gender and age bracket.

14.9 Summary

In this chapter, we learned how to predict the age and gender of a person in a photo by training two separate Convolutional Neural Networks. To accomplish this process, we trained our networks on the Adience dataset which includes labels for two genders along with eight separate age brackets. Our goal was to replicate the performance Levi et al. obtained on this dataset, where they obtained:

1. 50.57% exact and 84.70% one-off accuracy for age.
2. 86.8% exact accuracy for age.

We followed their CNN architecture while introducing (1) batch normalization layers and (2) additional dropout. During the training process, we applied additional data augmentation to help reduce overfitting. Overall, we were able to obtain:

1. **71.15%** exact and **88.28%** one-off accuracy for age.
2. **90.29%** exact accuracy for age.

Both of these results are *huge* improvements over all the original reported accuracies. After we evaluated our CNN performance, we then defined a custom Python script to predict the age and gender of people in photographs *not* a part of the Adience dataset. This task was accomplished by using a technique called *face aligning* before passing the ROI through the networks for classification.

It's worth noting that we *did not* use the same alignment tool as the Levi et al. Instead, we used our own (simpler) face alignment algorithm and achieved good results. When training your own

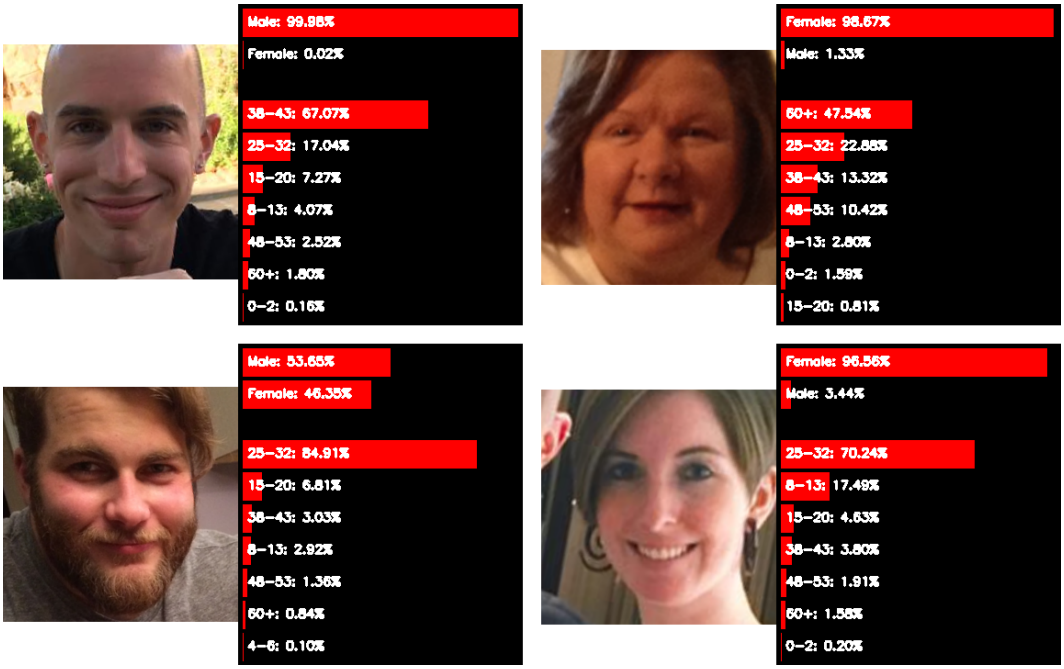


Figure 14.8: Example age and gender predictions for a sample of input images *outside* the Adience dataset.

custom age and gender prediction algorithms, I would suggest using the *same* face alignment tool for your training, validation, testing, and outside evaluation images – doing so will ensure that your images are pre-processed identically *and* that your model will generalize better.

15. Faster R-CNNs

Deep learning has impacted almost every facet of computer vision that relies on machine learning in a meaningful way. Image classification, image search engines (also known as Content-based Image Retrieval, or CBIR), localization and mapping (SLAM), image segmentation, to name a few, have all been changed since the latest resurgence in neural networks and deep learning. Object detection is no different.

One of the most popular deep learning-based object detection algorithms is the family of R-CNN algorithms, originally introduced by Girshick et al. in 2013 [43]. Since then, the R-CNN algorithm has gone under a number of iterations, improving the algorithm with each new publication, and outperforming traditional object detection algorithms (ex., Haar cascades [44]; HOG + Linear SVM [41]) at every step of the way.

In this chapter we'll discuss the Faster R-CNN algorithm and its components, including anchors, the base network, the Region Proposal Network (RPN), and Region of Interest (ROI) pooling. This discussion of Faster R-CNN building blocks will help you understand the core algorithm, how it works, and how end-to-end deep learning object detection is possible.

In the following chapter, we'll review the TensorFlow Object Detection API [45], including how to install it, how it works, and how to use the API to train your own Faster R-CNN object detectors on custom datasets.

These two chapters on Faster R-CNNs, along with the following two chapters on Single Shot Detectors (SSDs), will focus on object detection from a self-driving cars standpoint, demonstrating how to train object detectors to localize street signs and vehicles in images and video streams. You'll be able to use these discussions and code examples as a starting point for your own projects.

15.1 Object Detection and Deep Learning

Object detection, regardless of whether performed via deep learning or other computer vision techniques, has three primary goals — given an input image we wish to obtain:

1. **A list of bounding boxes**, or the (x, y)-coordinates for each object in an image
2. A **class label** associated with each bounding box
3. The **probability/confidence score** associated with each bounding box and class label

Inside Chapter 13 of the *Practitioner Bundle* we reviewed the traditional object detection pipeline, including:

- Sliding windows to localize objects at different locations
- Image pyramids, used to detect objects at varying scales
- Classification via a pre-trained CNN

In this manner we were able to frame object detection as classification, simply by utilizing sliding windows and image pyramids. The problems with this approach are numerous, but the primary ones include:

- **Slow and tedious:** running a sliding window at *every location* at every layer of an image pyramid is a time consuming process
- **Lack of aspect ratio:** Since our CNN requires a fixed-size input, we cannot encode the aspect ratio of a potential object into the ROI extracted and fed into the CNN. This leads to less accurate localizations.
- **Error prone:** Balancing speed (larger sliding windows steps and fewer image pyramid layers) with hopefully higher accuracy (more sliding window steps and more pyramid layers) is incredibly challenging. This issue is only further compounded by a lack of object aspect ratio in the ROI.

What we *really* need is an end-to-end deep learning-based object detector where we input an image to the network and obtain the bounding boxes and class labels for output. As we'll see, building an end-to-end object detector is not an easy task.

15.1.1 Measuring Object Detector Performance

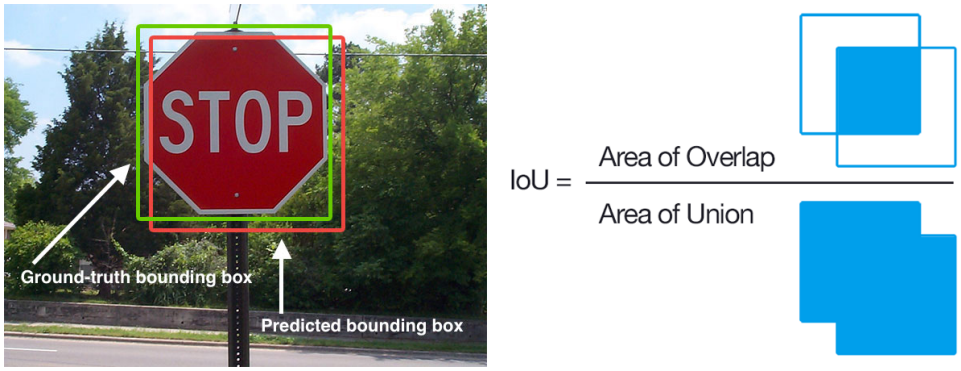


Figure 15.1: **Left:** An example of detecting a stop sign in an image. The predicted bounding box is drawn in red while the ground-truth bounding box is drawn in green. Our goal is to compute the Intersection of Union between these bounding box. **Right:** Computing the Intersection of Union is as simple as dividing the area of overlap between the bounding boxes by the area of union.

When evaluating object detector performance we use an evaluation metric called **Intersection over Union (IoU)**. You'll typically find IoU used to evaluate the performance of HOG + Linear SVM detectors [41], Convolutional Neural Network methods, such as Faster R-CNN [46], SSD [47], You Only Look Once (YOLO) [48, 49], and others; however, keep in mind that the actual algorithm used to generate the predicted bounding boxes does not matter.

Any algorithm that provides predicted bounding boxes (and optionally class labels) as output can be evaluated using IoU. More formally, in order to apply IoU to evaluate an arbitrary object detector, we need:

1. The *ground-truth bounding boxes* (i.e., the hand labeled bounding boxes from our testing set that specify where in an image our object is).

2. The *predicted bounding boxes* from our model.
3. If you want to compute recall along with precision, you'll also need the ground-truth class labels and predicted class labels.

As long as we have these two sets of bounding boxes we can apply IoU.

In Figure 15.1 (*left*) I have included a visual example of a ground-truth bounding box (green) versus a predicted bounding box (red). Computing IoU can therefore be determined by the equation illustration in Figure 15.1 (*right*).

Examining this equation you can see that IoU is simply a ratio. In the numerator we compute the **area of overlap** between the predicted bounding box and the ground-truth bounding box. The denominator is the **area of union**, or more simply, the area encompassed by both the predicted bounding box and ground-truth bounding box. Dividing the area of overlap by the area of union yields a final score — *the Intersection over Union*.

Where do the ground-truth examples come from?

Before we get too far, you might be wondering where the ground-truth examples come from. I've mentioned earlier in this chapter that our dataset must be "hand labeled", but what exactly does that mean?

When training your own object detector, you need a dataset. This dataset should be broken into (at least) two groups:

1. A training set used for training your object detector
2. A testing set for evaluating your object detector

You may also have a validation set used to tune the hyperparameters of your model.

Both the training and testing set will consist of:

1. The actual images themselves
2. The bounding boxes associated with the object(s) in the image. The bounding boxes are simply the (x, y)-coordinates of the object in the image.

The bounding boxes for the training and testing sets are *hand labeled*, hence why we call them the "ground-truth". Your goal is to take the training images + bounding boxes, construct an object detector, and then evaluate its performance on the testing set. **An IoU score > 0.5 is normally considered a "good" prediction.**

Why do we use Intersection over Union?

So far in this book we have primarily performed classification, where our model predicts a set of class labels for an input image — the predicted label with the highest probability is either *correct* or *incorrect*. This type of binary classification makes computing accuracy straightforward — it's either correct or not; however, for object detection, it's not so simple.

Figure 15.2: An example of computing Intersection over Unions for various bounding boxes. The more the predicted bounding box overlaps with the ground-truth bounding box, the better the prediction, and thus a higher IoU score.

In reality, it's *extremely unlikely* that the (x, y)-coordinates of our predicted bounding box are going to *exactly match* the (x, y)-coordinates of the ground-truth bounding box. Due to varying parameters of our model, such as layer used for feature extraction, anchor placement, loss function, etc., a complete and total match between predicted and ground-truth bounding boxes is simply unrealistic.

Because coordinates will not match *exactly*, we need to define an evaluation metric that rewards predicted bounding boxes for heavily overlapping with the ground-truth, as Figure 15.2 demonstrates.

In this illustration I have included three examples of good and bar IoU scores. Predicted bounding boxes that heavily overlap with the ground-truth bounding boxes have higher scores than those with less overlap. This behavior makes IoU an excellent metric for evaluating custom object detectors.

Again, we aren't concerned with an *exact* match of (x, y)-coordinates, but we do want to ensure that our predicted bounding boxes match as closely as possible — IoU is able to take this fact into account.

Implementing IoU by hand is outside the context of this book, although it is a fairly straightforward process. If you're interested in learning more about IoU, including a walkthrough of Python code demonstrating how to implement it, please see this PyImageSearch blog post: http://pyimg.co/ag4o5

Mean Average Precision (mAP)

In the context of machine learning, precision typically refers to accuracy — but in the context of object detection, IoU is our precision. However, we need to define a method to compute accuracy *per class* and across *all classes* in dataset. To accomplish this goal, we need **mean Average Precision (mAP)**.

To compute average precision for a single class, we determine the IoU of all data points for a particular class. Once we have the IoU we divide by the total class labels for that specific class, yielding the average precision. To compute the mean average precision, we compute the average IoU for all N classes — then we take the average of these N averages, hence the term mean average precision.

When reading object detection papers you'll typically see results reported in terms of mAP. This mAP value is derived by averaging the average precision on a per-class basis for all classes in the dataset. Typically we'll report **mAP@0.5**, indicating that in order for an object in the testing set to be marked as a "positive detection" it must have, at least, 0.5 IoU with the ground-truth. The 0.5 value is tunable, but 0.5 is fairly standard across most object detection datasets.

15.2 The (Faster) R-CNN Architecture

In this section we'll review the Faster R-CNN architecture. We'll start with a brief discussion on R-CNNs and how they have evolved over the series of three publications, leading to the Faster R-CNN architecture that we commonly use now. Finally, we'll examine the core building blocks of the Faster R-CNN architecture, first at a high level and then again in more detail.

As we'll see, the Faster R-CNN architecture is complex, with many moving parts — we'll focus our efforts on obtaining an understanding of tenets of this architecture prior to training the network on our own custom datasets in Chapter 16.

15.2.1 A Brief History of R-CNN

To better understand how the Faster R-CNN object detection algorithm works, we first need to review the history of the R-CNN algorithm, including its original incarnation and how it has evolved.

R-CNN

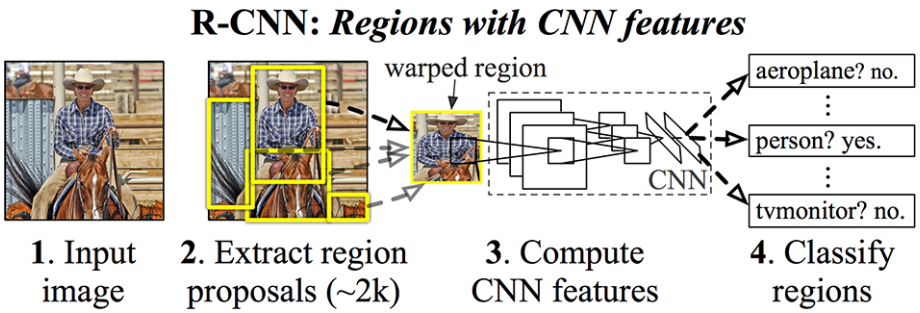


Figure 15.3: The original R-CNN architecture consisted of (1) accepting an input image, (2) extracting $\approx 2,000$ proposals via the Selective Search algorithm, (3) extracting features for each proposal ROI using a pre-trained CNN (such as one trained on ImageNet), and (4) classifying the features for each ROI using a class-specific Linear SVM. (Image credit: Figure 1 of Girshick et al. [43])

The first R-CNN paper, *Rich feature hierarchies for accurate object detection and semantic segmentation*, was published in 2013 by Girshick et al. [43] — we refer to this initial paper and associated implementation as simply R-CNN. An overview of the original R-CNN algorithm can be found in Figure 15.3, which includes a four step process:

- **Step #1:** Input an image
- **Step #2:** Extract regions proposals (i.e., regions of the image that potentially contain objects) using an algorithm such as Selective Search [50].
- **Step #3:** Use transfer learning, specifically feature extraction, to compute features for each proposal (which is effectively an ROI) using the pre-trained CNN.
- **Step #4:** Classify each proposal using the extracted features with a Support Vector Machine (SVM).

In the first step we input an image to our algorithm. We then run a region proposal algorithm as such Selective Search (or equivalent). The Selective Search algorithm *takes the place* of sliding windows and image pyramids, intelligently examining the input image at various scales and locations, thereby dramatically reducing the total number of proposal ROIs that will be sent to the network for classification. We can thus think of Selective Search as a smart sliding window and image pyramid algorithm.

A review of the Selective Search algorithm is outside the scope of this book so I recommend treating it as a "black box" that intelligently proposes ROI locations to you. For a detailed discussion of Selective Search refer to Uijlings et al. [50].

Once we have our proposal locations we crop each of them individually from the input image and apply transfer learning via feature extraction (Chapter 3 of the *Practitioner Bundle*). Instead of obtaining the final predictions from the CNN, we utilize feature extraction to enable a downstream classifier to learn more discriminating patterns from these CNN features.

The fourth and final step is to train a series of SVMs on top of these extracted features for each class.

Looking at this pipeline for the original R-CNN we can clearly see inspirations and parallels from traditional object detectors such as Dalal and Triggs seminal HOG + Linear SVM framework:

- Instead of applying an exhaustive image pyramid and sliding window, we are swapping in a more intelligent Selective Search algorithm
- Instead of extracting HOG features from each ROI, we're now extracting CNN features
- We're still training SVM(s) for the final classification of the input ROI, only we're training this SVM on the CNN features rather than the HOG ones

The primary reason this approach worked so well is due to the robust, discriminative features learned by a CNN — something we have explored thoroughly in Chapter 3 and Chapter 5 on transfer learning inside the *Practitioner Bundle*.

The problem with the original R-CNN approach is that it's still incredibly slow. And furthermore, we're not actually learning to localize via deep neural network.

Instead, we're leaving the localization to the Selective Search algorithm — we're only classifying the ROI once it's been determined as "interesting" and "worth examining" by the region proposal algorithm, which raises the question: is it possible to obtain end-to-end deep learning-based object detection?

Fast R-CNN

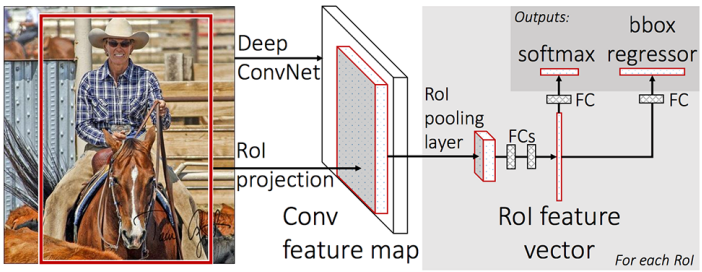


Figure 15.4: In the Fast R-CNN architecture, we still use Selective Search to obtain our proposals, but now we apply **ROI Pooling** by extracting a fixed-size window from the feature map and using these features to obtain the final class label and bounding box. (Image credit: Figure 1 of Girshick et al. [51])

Approximately a year and a half after Girshick et al. submitted the original R-CNN publication to arXiv, Girkshick published a second paper, *Fast R-CNN* [51]. Similar to the original R-CNN, Fast R-CNN algorithm still utilized Selective Search to obtain region proposals, but a novel contribution was made: **Region of Interest (ROI) Pooling**. A visualization of the new, updated architecture can be seen in Figure 15.4

In this new approach, we apply the CNN to the entire input image and extract a feature map from it using our network. ROI Pooling works by extracting a fixed-size window from the feature map and then passing it into a set of fully-connected layers to obtain the output label for the ROI. We'll discuss ROI Pooling in more detail in Section 15.2.5, but for the time being, understand that ROI Pooling operates over the feature map extracted from the CNN and extracts a *fixed-size window* from it.

The primary benefit here is that the network is now, effectively, end-to-end trainable:

1. We input an image and associated ground-truth bounding boxes
2. Extract the feature map
3. Apply ROI pooling and obtain the ROI feature vector
4. And finally use two sets of fully-connected layers to obtain (1) the class label predictions and (2) the bounding box locations for each proposal.

While the network is now end-to-end trainable, performance suffered dramatically at inference

(i.e., prediction) time by being dependent on the Selective Search (or equivalent) region proposal algorithm. To make the R-CNN architecture even *faster* we need to incorporate the region proposal *directly* into the R-CNN.

Faster R-CNN

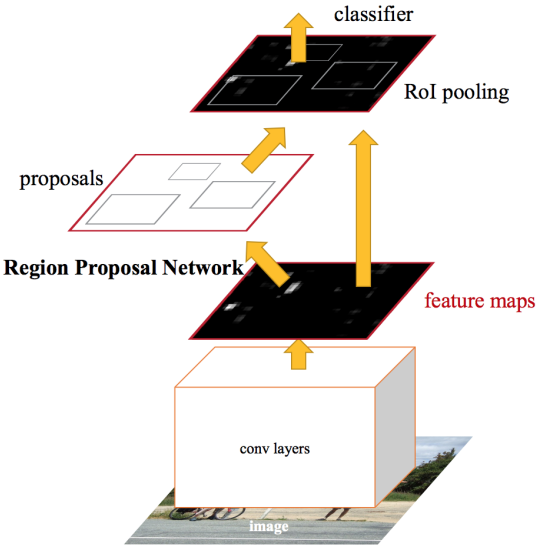


Figure 15.5: The latest incarnation of the R-CNN family, Faster R-CNN, introduces a **Region Proposal Network (RPN)** that bakes region proposal *directly* in the architecture, alleviating the need for the Selective Search algorithm. (Image credit: Figure 2 of Girshick et al. [46])

A little over a month after the Fast R-CNN paper was published, Girshick collaborated with Ren and Sun in a 2015 paper, *Faster R-NN: Towards Real-Time Object Detection with Region Proposal Networks* [46].

In this work, Girshick et al. created an additional component to the R-CNN architecture, a **Region Proposal Network (RPN)**. As the name of this module sounds, the goal of the RPN is to remove the requirement of running Selective Search prior to inference and instead bake the region proposal *directly* into the R-CNN architecture.

Figure 15.5 provides a visualization of the updated architecture. Here we can see an input image is presented to the network and its features extracted via pre-trained CNN (i.e., the base network). These features, in parallel, are sent to two different components of the Faster R-CNN architecture.

The first component, the RPN, is used to determine *where* in an image a potential object could be. At this point we do not know *what* the object is, just that there is *potentially* an object at a certain location in the image.

The proposed bounding box ROIs are based on the Region of Interest (ROI) Pooling module of the network along with the extracted features from the previous step. ROI Pooling is used to extract fixed-size windows of features which are then passed into two fully-connected layers (one for the class labels and one for the bounding box coordinates) to obtain our final localizations.

We'll discuss the RPN in detail inside Section 15.2.4, but in essence, we are now going to place **anchors** spaced uniformly across the entire image at varying scales and aspect ratios. The RPN will then examine these anchors and output a set of proposals as to where it "thinks" an object exists.

It's important to note here that the RPN is not actually labeling the ROI; instead, it's computing its "objectness score" and asking: *"Does this region look like an object of some sort?"* I personally like to think of the RPN and objectness score as a binary classifier of sorts where the RPN is labeling each ROI as "background" or "foreground". If the RPN thinks the ROI is "foreground" then the ROI is worth further consideration by the ROI Pooling and final label + bounding box fully-connected layers.

At this point the *entire* architecture is end-to-end trainable and the complete object detection pipeline takes place inside the network, including:

1. Region proposal
2. Feature extraction
3. Computing the bounding box coordinates of the objects
4. Providing class labels for each bounding box

And furthermore, depending on which GPU and base architecture is used, it's now possible to obtain $\approx 7 - 10$ frames per second (FPS), a huge step towards making real-time object detection with deep learning a reality.

Now that we have a brief introduction to the components in a (Faster) R-CNN architecture, let's examine each of them individually in more detail.

15.2.2 The Base Network

In Figure 15.5 we can see the general modules in the Faster R-CNN architecture. After we input the image to the architecture, the first component we come across is the base network. The base network is typically a CNN pre-trained for a particular classification task. This CNN will be used for transfer learning, in particular, feature extraction.

In the context of object detection, we typically use a CNN pre-trained on the ImageNet dataset. We use a pre-trained CNN here as the features learned by a particular layer are often transferrable to classification tasks outside what the original network was trained on. For a complete review on transfer learning, feature extraction, fine-tuning, and how we can use transfer learning via pre-trained networks, please refer to *Chapter 3* of the *Practitioner Bundle*.

The original Faster R-CNN paper used VGG [17] and ZF [52] as the base networks. Today, we would typically swap in a deeper, more accurate base network such as ResNet [21, 22], or a smaller, more compact network for resource contained devices, such as MobileNet [53].

One important aspect of object detection networks is that they should be *fully-convolutional*, not to be confused with fully-connected. A fully-convolutional neural network does not contain the fully-connected layers typically found at the end of a network prior to making output predictions. In the context of *image classification*, removing the fully-connected layers is normally accomplished by applying average pooling across the entire volume prior to a single dense softmax classifier used to output the final predictions

A fully-convolutional neural network enjoys two primary benefits, including being:

1. Fast, due to all convolution operations
2. Able to accept images of any spatial resolution (i.e., width and height), provided that the image and network can fit into memory, of course

When implementing object detection with deep learning it's important that we do not rely on fixed-size input images that require us to force the image to a specific dimension. Not only can fixed-sizes distort the aspect ratio of the input image, but it has the even worse side effect of making it extremely challenging for the network to detect small objects — if we reduce the spatial dimensions of our image too much, small objects will now appear as tiny pixel blobs, too small for the network to detect. Therefore, it's important that we allow the network to accept arbitrary spatial resolution images and allow input image size to be a decision made by the network developer or engineer.

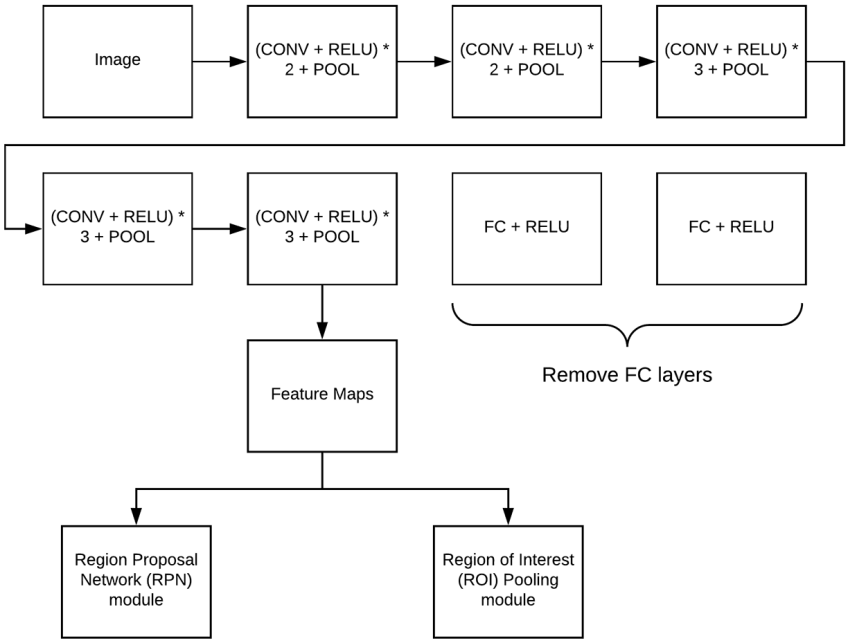


Figure 15.6: The base network is used to extract features from the input image. We ensure our network is fully-convolutional by removing the fully-connected layers at the end of the network and instead utilizing the output of the CONV and POOL layers. This process allows our network to handle input images of arbitrary spatial dimensions. The features are then fed into the RPN and ROI Pooling modules.

However, in the case of VGG in Girshick et al., we know there are fully-connected layers at the end of VGG — it is thus not a fully-convolutional network. There is a way to change the behavior of the CNN though. Keep in mind that we only need the output of a specific CONV (or POOL) layer in the network — this output is our feature map, the process of which is visualized in Figure 15.6.

We can obtain this feature map by propagating the input image through the network and stopping at our target layer (we do not have to pass the image through the entire network to obtain our feature map). The fact that VGG originally only accepted a 224×224 input image is entirely arbitrary now that we're only interested in the output of a specific CONV + POOL layer. This feature map will be used by Region Proposal Network and ROI Pooling module later in the Faster R-CNN architecture.

15.2.3 Anchors

In traditional object detection pipelines we would use either (1) a combination of a sliding window + image pyramid or (2) a Selective Search-like algorithm to generate proposals for our classifier. Since our goal is to develop an end-to-end object detector using deep learning that includes the proposal module, we need to define a method that will generate our proposal ROIs.

The core separation between classification and object detection is the prediction of bounding boxes, or (x, y)-coordinates surrounding an object. Thus, we might expect our network to return a tuple consisting of the bounding box coordinates of a particular object. But, there is a problem with this approach, namely:

1. How do we handle a network predicting values *outside* the boundaries of the image?
2. How do we encode restrictions such as $x_{min} < x_{max}$ and $y_{min} < y_{max}$?

It sounds out that this is a near impossible problem to solve. However, the solution proposed by

Girchick et al., called **anchors**, is a clever and novel one.

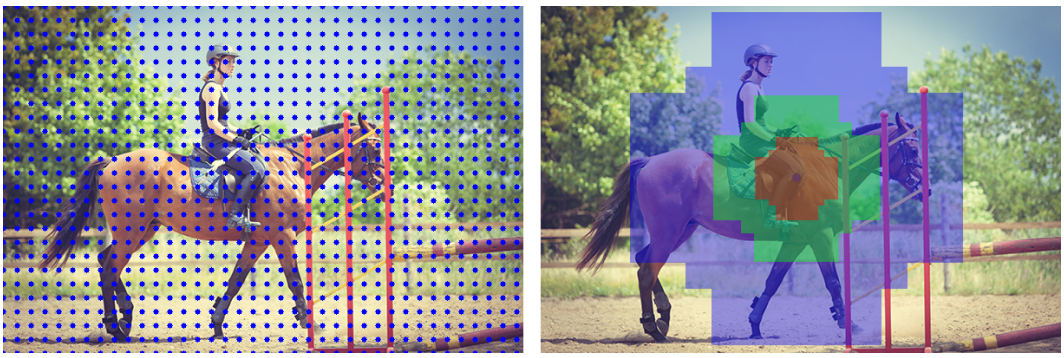


Figure 15.7: **Left:** Creating anchors starts with the process of sampling the coordinates of an image every r pixels ($r = 16$ in the original Faster R-CNN implementation). **Right:** We create a total of nine anchors centered around *each* sampled (x, y)-coordinate. In this visualization, $x = 300, y = 200$ (center blue coordinate). The nine total anchors come from every combination of scale: 64×64 (red), 128×128 (green), 256×256 (blue); and aspect ratio: $1:1$, $2:1$, $1:2$.

Instead of trying to predict the raw (x, y)-coordinates of the bounding boxes, we can instead learn to predict their offsets from the reference boxes, namely: $\Delta_{x-center}$, $\Delta_{y-center}$, Δ_{width}, and Δ_{height}. These delta values allow us to obtain a better fit to our reference box without having to predict the actual raw (x, y)-coordinates, enabling us to bypass the potentially impossible problem of encoding bounding box "rules" into the network.

So where do these reference bounding boxes come from? We need to generate the anchors ourselves without utilizing a Selective Search algorithm. To accomplish this process, we first need to uniformly sample points across an input image (Figure 15.7, *left*). Here we can see an input image that is 600×400 pixels — we have labeled each point at a regularly sampled integer (at an interval of sixteen pixels) with a blue circle.

The next step is to create a set of anchors at each of the sampled points. As in the original Faster R-CNN publication, we'll generate nine anchors (which are fixed bounding boxes) with varying sizes and aspect ratios surrounding a given sampled point.

The colors of the bounding boxes are our scales/sizes, namely: 64×64, 128×128, and 256×256. For each scale we also have the aspect ratio, $1:1$, $1:2$, and $2:1$. Each combination of scale and aspect ratio yields nine total anchors. This combination of scale and aspect ratio yields us considerable coverage over all possible object sizes and scales in the input image (Figure 15.7, *right*).

However, there is a problem here once we break down the total number of anchors generated:

- If we use a stride of 16 pixels (the default for Faster R-CNN) on a 600×800 image, we'll obtain a total of 1,989 total positions.
- With nine anchors surrounding each of the 1,989 positions, we now have a total of $1,989 \times 9 = 17,901$ bounding box positions for our CNN to evaluate.

If our CNN classified each of the 17,901 bounding boxes our network would be only slightly faster than exhaustively looping over each combination of sliding window and image pyramid. Luckily, with the Region Proposal Network (RPN) we can dramatically reduce the number of candidate proposal windows, leaving us with a much more manageable size.

15.2.4 Region Proposal Network (RPN)

If the goal of generating anchors is to obtain good coverage over all possible scales and sizes of objects in an image, the goal of the **Region Proposal Network (RPN)** is to prune the number of generated bounding boxes to a more manageable size.

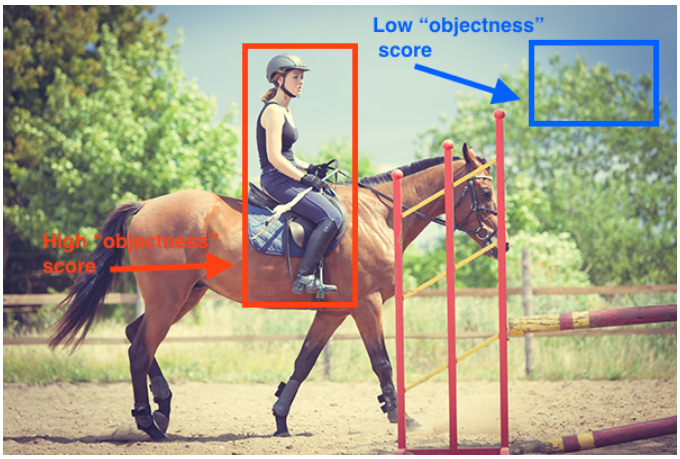


Figure 15.8: The goal of the Region Proposal Module is to accept a set of anchors and quickly determine the "objectness" of a region in image. Namely, this involves labeling the ROI as either *foreground* or *background*. Background anchor regions are discarded while foreground objects are propagated to the ROI Pooling module.

The RPN module is simplistic yet powerful, consisting of two outputs. The top of the RPN module accepts an input, which is our convolutional feature map from Section 15.2.2. We then apply a 3×3 `CONV`, learning 512 filters.

These filters are fed into two paths in parallel. The first output (*left*) of the RPN is a score that indicates whether the RPN thinks the ROI is *foreground* (worth examining further) or *background* (discard). Figure 15.8 provides a visualization of labeling the "objectness" of an input ROI.

Again, the RPN is not actually labeling the ROI — it's just trying to determine if the ROI is either *background* or *foreground*. The actual labeling of the ROI will take place later in the architecture (Section 15.2.6). The dimensionality of this output is $2 \times K$ where K is the total number of anchors, one output for the foreground probability and the second output for the background probability.

The second output (*right*) is our bounding box regressor used to adjust the anchors to better fit the object that it is surrounding. Adjusting the anchors is again accomplished via 1×1 convolution, but this time outputting a $4 \times K$ volume. The output is $4 \times K$ as we are predicting the four delta (i.e., offset) values: $\Delta_{x-center}$, $\Delta_{y-center}$, Δ_{width}, Δ_{height}.

Provided that our foreground probability is sufficiently large, we then apply:

- Non-maxima suppression to suppress overlapping
- Proposal selection

There will naturally be many overlapping anchor locations as per Section 15.2.3 above — non-maxima suppression helps reduce the number of locations to pass on to the ROI Pooling module. We further reduce the number of locations to pass into the ROI pooling module via proposal selection Here we take only the top N proposals and discard the rest.

In the original Faster R-CNN publication, Girshick et al. set $N = 2,000$, but we can get away with a much smaller N, such as $N = 10, 50, 100, 200$ and still obtain good predictions.

The next step in the pipeline would be to propagate the ROI and deltas to the Region of Interest (ROI) Pooling module (Section 15.2.5), but let's first discuss how we might train the RPN.

Training the RPN

During training, we take our anchors and put them into two different buckets:

- **Bucket #1 — Foreground:** All anchors that have a 0.5 IoU with a ground-truth object bounding box.
- **Bucket #2 — Background:** All anchors that have < 0.1 IoU with a ground-truth object.

Based on these buckets we randomly sample between the two to maintain an equal ratio between background and foreground.

From there, we need to consider our loss functions. In particular, the RPN module has two loss functions associated with it. The first loss function is for classification which measures the accuracy of the RPN predicting foreground vs. background (binary cross-entropy works nicely here).

The second loss function is for our bounding box regression. This loss function only operates on the foreground anchors as backgrounds anchors would have no sense of a bounding box (and we should have already detected "background" and discarded it).

For the bounding box regression loss function, Girshick et al. utilized a variant or L1 loss called **Smooth L1 loss**. Since it's unrealistic for us to 100% accurately predict the ground-truth coordinates of a bounding box (addressed in Section 15.1.1), Smooth L1 loss allows bounding boxes that are "sufficiently close" to their corresponding ground-truth boxes to be essentially correct and thereby diminish the impact of the loss.

15.2.5 Region of Interest (ROI) Pooling

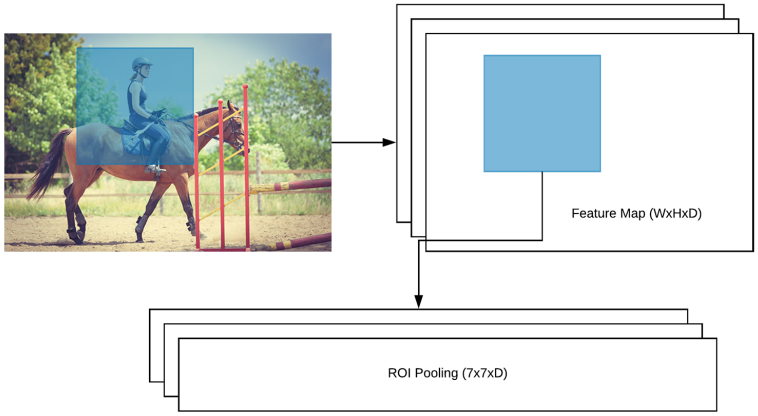


Figure 15.9: The goal of the Region of Interest ROI Pooling module is to take all N proposal locations and then extract the corresponding ROI features from the convolutional feature map. The ROI Pooling module then resizes the dimensions of the extracted features for the ROI down to $7 \times 7 \times D$ (where D is the depth of the feature map). This fixed size prepares the feature for the two upcoming `FC` layers in the next module.

The goal of the ROI Pooling module is to accept all N proposal locations from the RPN module and crop out feature vectors from the convolutional feature map in Section 15.2.2 (Figure 15.9). Cropping feature vectors is accomplished by:

- Using array slicing to extract the corresponding patch from the feature map
- Resizing it to $14 \times 14 \times D$ where D is the depth of the feature map
- Applying a max pooling operation with 2×2 strides, yielding a $7 \times 7 \times D$ feature vector.

The final feature vector obtained from the ROI Pooling module can now be fed into the Region-based Convolutional Neural network (covered in the next section) which we will use to obtain the final bounding box coordinates for each object along with the corresponding class label.

For more information on how ROI Pooling is implemented, please see the original Girshick et al. publication [46] as well as the excellent tutorial, Faster R-CNN: Down the Rabbit Hole of Modern Object Detection [54].

15.2.6 Region-based Convolutional Neural Network

The final stage in our pipeline is the Region-based Convolutional Neural Network, or as we know it, R-CNN. This module serves two purposes:

1. Obtain the final class label predictions for each bounding box location based on the cropped feature map from the ROI Pooling module
2. Further refine the bounding box prediction (x, y)-coordinates for better prediction accuracy

In practice, the R-CNN component is implemented via two fully-connected layers, as Figure 15.10 demonstrates. On the far *left* we have the input to our R-CNN, which are the feature vectors obtained from the ROI Pooling module. These features pass through two fully-connected layers (each 4096-d) before being passed into the final two `FC` layers which will yield our class label (or background class) along with the bounding box delta values.

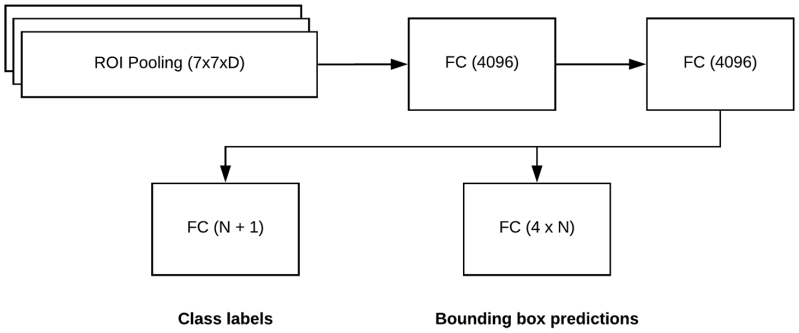


Figure 15.10: The final module in the Faster R-CNN architecture takes the output of the ROI Pooling component and passes it through two `FC` layers (each 4096-d), similar to the final layers of classification networks trained on ImageNet. The output of these `FC` layers feed into the final two `FC` layers in the network. One `FC` layer is $N+1$—d, a node for each of the class labels plus an additional label for the background. The second `FC` layer is $4 \times N$ which represents the deltas for the final predicted bounding boxes.

One `FC` layer has $N+1$ nodes, where N is the total number of class labels. The addition of the extra dimension is used to indicate the background class, just in case our RPN module let a background region through.

The second `FC` layer is $4 \times N$. The N here is again our total number of class labels. The four values are our corresponding deltas for $\Delta_{x-center}$, $\Delta_{y-center}$, Δ_{width}, Δ_{height} which will be transformed to our final bounding boxes.

These two outputs again imply that we'll have two loss functions:

1. Categorical cross-entropy for classification
2. Smooth L1 loss for bounding box regression

We use *categorical* cross-entropy rather than *binary* cross-entropy here as we are computing probabilities for each of our N classes versus the binary case (background vs. foreground) in the RPN module. The final step is to apply non-maxima suppression class-wise to our set of bounding

boxes — these bounding boxes and associated class labels are considered the final predictions from our network.

15.2.7 The Complete Training Pipeline

We have a choice to make when training the entire Faster R-CNN pipeline. The first choice is to train the RPN module, obtain satisfiable results, and then move on to training the R-CNN module. The second choice is to combine the four loss functions (two for the RPN module, two for the R-CNN module) via weighted sum and then *jointly* train all four. Which one is better?

In nearly all situations you'll find that jointly training the entire network end-to-end by minimizing the weighted sum of the four loss functions not only *takes less time* but also *obtains higher accuracy* as well.

In our next chapter we'll learn how to use the TensorFlow Object Detection API to train our own Faster R-CNN networks. If you're interested in more details on the Faster R-CNN pipeline, RPN and ROI Pooling modules, along with additional notes on how we jointly minimize the four loss functions, be sure to refer to the original Faster R-CNN publication [46] as well as the TryoLabs article [54].

15.3 Summary

In this chapter we started by reviewing the Faster R-CNN architecture, along with its earlier variants, by Girshick et al. [43, 46, 51]. The R-CNN architecture has gone under a few iterations and improvements, but with the latest Faster R-CNN architecture we are able to train end-to-end deep learning object detectors.

The architecture itself includes four primary components. The first component is the **base network** (i.e., ResNet, VGGNet, etc.) which is used as a feature extractor.

We then have the **Region Proposal Network (RPN)**, which accepts a set of **anchors**, and outputs proposals as to where it thinks objects are in an image. It's important to note that the RPN does not know what the object is in the image, just that a potential object exists at a given location.

Region of Interest Pooling is used to extract feature maps from each proposal region.

Finally a **Region-based Convolutional Neural Network** is used to (1) obtain the final class label predictions for the proposal and (2) further refine the proposal locations for better accuracy.

Given the large number of moving parts in the R-CNN architecture, it is *not* advisable to implement the entire architecture by hand. Instead, it's recommended to use existing implementations such as the TensorFlow Object Detection API [55] or Luminoth from TryoLabs [56].

In our next chapter we will learn how to train a Faster R-CNN using the TensorFlow Object Detection API to detect and recognize common United States street/road signs.

16. Training a Faster R-CNN From Scratch

In our previous chapter we discussed the Faster R-CNN architecture and its many components, including the Region Proposal Network (RPN) and ROI Pooling module. While the previous chapter focused strictly on the inner-working parts, this chapter will switch towards implementation and how to actually *train* a Faster R-CNN on a custom dataset.

We are going to be taking a "deep learning for self-driving cars" approach to both this chapter and Chapter 18 on Single Shot Detectors. By the end of this chapter you'll be able to:

1. Install and configure the TensorFlow Object Detection API on your system
2. Build an image dataset + image annotations in the TensorFlow Object Detection API Form
3. Train a Faster R-CNN on the LISA Traffic Signs dataset (or your own custom dataset)
4. Evaluate the accuracy and apply the trained Faster R-CNN to input images and videos

16.1 The LISA Traffic Signs Dataset

The LISA Traffic Signs dataset was curated and put together by Mogelmose et al. and fully detailed in their 2012 paper, *Vision-Based Traffic Sign Detection and Analysis for Intelligent Driver Assistance Systems: Perspectives and Survey* [57].

The dataset consists of 47 different United States traffic sign types, including stop signs, pedestrian crossing signs, etc. An example image from the LISA Traffic Signs dataset can be seen in Figure 16.1. The dataset was initially captured via video but the individual frames and associated annotations are also included.

There are a total of 7,855 annotations on 6,610 frames. Road signs vary in resolution, from 6×6 to 167×168 pixels. Furthermore, some images were captured in a lower resolution 640×480 camera while others were captured on a higher resolution 1024×522 pixels. Some images are grayscale while others are color. The variance in camera quality and capture color space make this an interesting dataset to study in terms of object detection.

The full LISA Traffic Signs dataset is over 7GB, so in order to train our network faster (and learn the fundamentals of object detection quicker), we will be sampling three traffic sign classes: stop sign, pedestrian crossing, and signal ahead signs. We'll be training our Faster R-CNN on this

Figure 16.1: The LISA Traffic Signs datasets consists of 47 different United States traffic signs with 7,855 annotations over 6,610 frames. Here we can see an example image containing two stop signs. Our goal will be to detect and label traffic signs such as this stop sign.

three class sampled dataset but you are welcome to train on the full dataset as well (but I would recommend replicating our results in this chapter *before* you make any changes).

The LISA Traffic Signs homepage can be found using this link: http://pyimg.co/lp6xo. From there you'll want to scroll to the bottom of the page to the "Access" section and download the `.zip` archive of the dataset. I have chosen to store the dataset in a directory appropriately named `lisa`. Once the download is complete, unarchive it:

```
$ mkdir lisa
$ cd lisa
$ mv signDatabasePublicFramesOnly.zip ./
$ unzip signDatabasePublicFramesOnly.zip
```

16.2 Installing the TensorFlow Object Detection API

When putting this book together I evaluated *many* deep learning-based object detection implementations, including pure Keras-based libraries, mxnet-based packages, Caffe implementations, and even Torch libraries.

Object detection is not only much harder to *train* a network on, but significantly more challenging to *implement* as well, as there are many more components, some of which require custom layers and loss functions. After reading Chapter **??** on the fundamentals of Faster R-CNNs, it should be clear there are many modules that would need to be implemented by hand.

Implementing the entire Faster R-CNN architecture is not something that can be covered in this book, for a number of reasons, including:

1. Implementing and explaining the Faster R-CNN architecture by hand using the same style used throughout the rest of the book (code blocks and detailed explanations) would take hundreds (if not more) of pages.
2. Object detection libraries and packages tend to be fragile in their nature as custom layers and loss methods are used. When a new version of their associated backend library is released, the risk of breaking such a custom module is high.

The goal here is to instead discuss a library that is least fragile as possible, giving you the best of both worlds: knowledge of the Faster R-CNN architecture and ability to train the network from scratch.

Because of the fragility of custom, ground-up implementations, we are going to use the TensorFlow Object Detection API (TFOD API) in this chapter and Chapter 18 on Single Shot Detectors. Not only is TensorFlow one of the most used deep learning libraries, but the TFOD API is supported by Google itself — there is a large community of both open source, research, and corporate developers working on the project, helping reduce the fragile nature of the library.

If you're interested in learning more about the TFOD API, be sure to take a look at their official GitHub page [45] as well as their 2017 CVPR paper, *Speed/accuracy trade-offs for modern convolution object detectors* [55].

To install and configure the TensorFlow Object Detection API, please refer to the following page of the companion website for this book: http://pyimg.co/7b7xm

If you do not have access to the companion website, please refer to the first few pages of this book for the companion website registration link.

Using the companion website for the TFOD API install instructions ensures that they can always be kept up to date, something that a printed book cannot guarantee. Follow the instructions using the link above to configure your machine and from there we can train your first Faster R-CNN!

16.3 Training Your Faster R-CNN

The first part of this section will detail and review project structure. It's important that you understand how to properly structure your project, otherwise you may run into hard to diagnose errors during training or evaluation.

The TensorFlow API requires our images and annotations be in a particular record format — we'll discuss how to take an image dataset + associated annotations and put them in the required format. From there we can train our Faster R-CNN architecture, evaluate its performance, and apply it to sample testing images.

16.3.1 Project Directory Structure

Between the TFOD API and our custom scripts used to build our input dataset, train a network, evaluate performance, and apply our models to custom images, there are a number of moving parts. Furthermore, some of these scripts live solely inside the TFOD API directory structure while other scripts will need to be hand-implemented by us.

Let's start by listing the TFOD API Python scripts we'll be using. My TFOD API was cloned into my home directory, so my root path to the TFOD API is `/home/adrian/models/research`:

```
$ cd ~/models/research/
$ pwd
/home/adrian/models/research
$ ls -l
adversarial_crypto
adversarial_text
adv_imagenet_models
...
object_detection
...
```

Your TFOD root path may be different depending on where you cloned it.

There are quite a number of subdirectories here, but the one we are most interested in is `object_detection`:

```
$ ls -A1 object_detection/*.py
object_detection/eval.py
object_detection/evaluator.py
object_detection/eval_util.py
object_detection/exporter.py
object_detection/exporter_test.py
object_detection/export_inference_graph.py
object_detection/__init__.py
object_detection/trainer.py
object_detection/trainer_test.py
object_detection/train.py
```

We'll be using three of these Python scripts, including:

1. `train.py`: Used to train our deep learning-based object detectors.
2. `eval.py`: Runs concurrently with `train.py` to evaluate on the testing set as we train.
3. `export_inference_graph.py`: After our model is fully trained we'll need to freeze the weights and export the model so we can import it into our own applications.

Again, these three scripts are part of the TFOD API and are *not* part of any custom code or implementation we will be writing. The directory structure and project structure for our Faster R-CNN custom scripts and training/evaluation configurations can be seen below:

```
| --- ssds_and_rcnn
|    |--- build_lisa_records.py
|    |--- config
|    |    |--- __init__.py
|    |    |--- lisa_config.py
|    |--- lisa/
|    |    |--- allAnnotations.csv
|    |    |--- categories.txt
...
|    |    |--- vid8/
|    |    |--- vid9/
|    |    |--- videoSources.txt
|    |--- predict.py
|    |--- predict_video.py
```

There are a number of configurations required to build our custom object detector so we'll create a file named `lisa_config.py` to store any configurations, such as file paths, class labels, etc.

The `build_lisa_records.py` will be used to accept an input set of images, creating the training and testing splits, and then create TensorFlow compatible record files that can be used for training.

After our network has been trained, we can use both `predict.py` and `predict_video.py` to apply our network to input images or video files/streams, respectively.

The `setup.sh` script contains a single command that can be used to easily update your `PYTHONPATH` to include the TFOD API imports when you execute your Python scripts via the command line.

The `lisa` directory contains all files relevant to the LISA Traffic Signs dataset such as the raw images themselves and the serialized record files. I have decided to store the LISA dataset on a separate hard drive on my system (where I have more space) and then create a sym-link to the dataset in my project structure, similar to what we have done in previous chapters of this

book. Again, this is my personal preference but you may have your own best practices that you are comfortable with.

Inside the LISA dataset directory I would also suggest creating the following directories for your experiments and data:

```
$ cd lisa
$ mkdir records experiments
$ mkdir experiments/training experiments/evaluation experiments/exported_model
```

The `records` directory will store three files:

1. `training.record`: The serialized image dataset of images, bounding boxes, and labels used for training.
2. `testing.record`: The images, bounding boxes, and labels used for testing.
3. `classes.pbtxt`: A plaintext file containing the names of the class labels and their unique integer IDs.

We then have an `experiments` subdirectory that will house any files used for our training experiments. Inside the `experiments` directory there are three additional subdirectories: `training`, `evaluation`, and `exported_model`.

The `training` directory will store the special pipeline configuration file used to instruct the TFOD API how to train our model, the pre-trained model we'll be using for fine-tuning, and any model checkpoints created during training.

While we train our network we'll also be running an evaluation script provided by the TFOD API — any logs generated by the evaluation script will be stored in the `evaluation` directory, enabling us to use TensorFlow tools to graph and plot the training process using the special TensorBoard utility.

Finally, `exported_model` will store the final exported, frozen weight model after training has completed.

When I first started using the TFOD API, I was a bit overwhelmed by the number of files, directories, and subdirectories required in order to use it. Through trial and error I developed this proposed directory structure as it properly balances ease of use with what the TFOD API requires.

For your convenience, I have included a template of the directory structure in the code downloads associated with this book. Feel free to modify this proposed structure as you see fit, but until you successfully train your first network, consider using my proposed method.

16.3.2 Configuration

The first file we're going to examine is the `setup.sh` file:

```
#!/bin/sh

export PYTHONPATH=$PYTHONPATH:/home/adrian/
    models/research:/home/adrian/models/research/slim
```

All this file is doing is updating our `PYTHONPATH` variable to include the TFOD API imports.

You'll want to update these paths to point to where you cloned the TFOD API repository on your own system. *Do not* copy and paste my file paths. Additionally, the `PYTHONPATH` should be on a *single* line — I needed to use two lines (due to text width restrictions of the book) to fit the entire command.

If we *do not* update our PYTHONPATH to include the TFOD API, our TensorFlow imports will fail when executing our scripts. Whenever I am working with the TFOD API, I always open up a shell and source the file:

```
$ source setup.sh
```

Sourcing the setup.sh script and update my PYTHONPATH to include the TFOD API files ensures that my TensorFlow imports will succeed.

Next, let's define the lisa_config.py configuration file:

```
1  # import the necessary packages
2  import os
3
4  # initialize the base path for the LISA dataset
5  BASE_PATH = "lisa"
6
7  # build the path to the annotations file
8  ANNOT_PATH = os.path.sep.join([BASE_PATH, "allAnnotations.csv"])
```

Lines 5 defines the BASE_PATH to the LISA Traffic Signs directory (i.e., where the raw images, records, and experiment logs live). Here, the lisa directory will live in the same directory as our build_lisa_records.py and predict.py scripts.

Line 8 uses the BASE_PATH to derive the path to the annotations file provided with the LISA Traffic Signs dataset download.

Using the BASE_PATH we can also derive the path to our output training and testing records (75% for training, 25% for testing), along with the class labels plaintext file:

```
10 # build the path to the output training and testing record files,
11 # along with the class labels file
12 TRAIN_RECORD = os.path.sep.join([BASE_PATH,
13      "records/training.record"])
14 TEST_RECORD = os.path.sep.join([BASE_PATH,
15      "records/testing.record"])
16 CLASSES_FILE = os.path.sep.join([BASE_PATH,
17      "records/classes.pbtxt"])
18
19 # initialize the test split size
20 TEST_SIZE = 0.25
21
22 # initialize the class labels dictionary
23 CLASSES = {"pedestrianCrossing": 1, "signalAhead": 2, "stop": 3}
```

When training our Faster R-CNN, we'll only be interested in three classes:

1. Pedestrian crossings
2. Signal ahead
3. Stop signs

We could certainly train on all 47 traffic sign classes included with LISA, but for the sake of this example we'll only train on three classes, enabling us to train the network faster and examine the results. I will leave training on the *full* LISA dataset as an exercise to you.

16.3.3 A TensorFlow Annotation Class

When working with the TFOD API, we need to build a dataset consisting of both the images and their associated bounding boxes. But before we can get to building the dataset, we need to consider what makes up "data point" for object detection? According to the TFOD API, we need to supply a number of attributes, including:

- The TensorFlow-encoded image
- The width and height of the image
- The file encoding of the image (i.e., JPG, PNG, etc.)
- The filename
- A list of bounding box coordinates, normalized in the range $[0, 1]$, for the image
- A list of class labels for each bounding box
- A flag used to encode if the bounding box is "difficult" or not (you'll almost always want to leave this value as "0", or "not difficult" so TensorFlow trains on it — the difficult flag is a remnant of the VOC challenge [58]).

Encoding all of this information in a object requires quite a bit of code, so in order to:

1. Keep our build script neat and tidy
2. Reuse code, and therefore reduce the potential of inserting bugs in our code

We're going to build a `TFAnnotation` class to encapsulate encoding an object detection data point in TensorFlow format. Our `TFAnnotation` class will live in `tfannotation.py` insider the `utils` sub-module of `pyimagesearch`:

```
| --- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- io
...
|    |--- utils
|    |    |--- __init__.py
|    |    |--- agegenderhelper.py
...
|    |    |--- tfannotation.py
```

Open up `tfannotation.py` and insert the following code:

```
1  # import the necessary packages
2  from object_detection.utils.dataset_util import bytes_list_feature
3  from object_detection.utils.dataset_util import float_list_feature
4  from object_detection.utils.dataset_util import int64_list_feature
5  from object_detection.utils.dataset_util import int64_feature
6  from object_detection.utils.dataset_util import bytes_feature
```

Lines 2-6 import TensorFlow dataset utility functions used to serialize and encode various Python attributes and objects — we'll be using these functions to encode integers, strings, lists, etc. We can then define the constructor to `TFAnnotation`:

```
8  class TFAnnotation:
9      def __init__(self):
10         # initialize the bounding box + label lists
11         self.xMins = []
12         self.xMaxs = []
```

```
13            self.yMins = []
14            self.yMaxs = []
15            self.textLabels = []
16            self.classes = []
17            self.difficult = []
18
19            # initialize additional variables, including the image
20            # itself, spatial dimensions, encoding, and filename
21            self.image = None
22            self.width = None
23            self.height = None
24            self.encoding = None
25            self.filename = None
```

As we can see, our constructor is simply performing a series of initializations. The `xMins`, `xMaxs`, `yMins`, and `yMaxs` will store the (x, y)-coordinates for our bounding boxes, respectively.

The `textLabels` list is a list of human readable class labels for each bounding box. Similarly, we have `classes`, a list of integer IDs for each class label.

Line 21 will store the TensorFlow-encoded image, along with its width, height, image encoding type, and filename.

As we'll see in Section 16.3.4 below, we'll first instantiate the `TFAnnotation` object and then set each of these attributes. When we are ready to build the actual TensorFlow data point, we can call the `build` method:

```
27      def build(self):
28            # encode the attributes using their respective TensorFlow
29            # encoding function
30            w = int64_feature(self.width)
31            h = int64_feature(self.height)
32            filename = bytes_feature(self.filename.encode("utf8"))
33            encoding = bytes_feature(self.encoding.encode("utf8"))
34            image = bytes_feature(self.image)
35            xMins = float_list_feature(self.xMins)
36            xMaxs = float_list_feature(self.xMaxs)
37            yMins = float_list_feature(self.yMins)
38            yMaxs = float_list_feature(self.yMaxs)
39            textLabels = bytes_list_feature(self.textLabels)
40            classes = int64_list_feature(self.classes)
41            difficult = int64_list_feature(self.difficult)
```

The above code block calls TensorFlow's encoding functions on each of our attributes. We can then build a `data` dictionary from these encoded objects:

```
43            # construct the TensorFlow-compatible data dictionary
44            data = {
45                  "image/height": h,
46                  "image/width": w,
47                  "image/filename": filename,
48                  "image/source_id": filename,
49                  "image/encoded": image,
50                  "image/format": encoding,
51                  "image/object/bbox/xmin": xMins,
52                  "image/object/bbox/xmax": xMaxs,
```

```
53                 "image/object/bbox/ymin": yMins,
54                 "image/object/bbox/ymax": yMaxs,
55                 "image/object/class/text": textLabels,
56                 "image/object/class/label": classes,
57                 "image/object/difficult": difficult,
58             }
59
60             # return the data dictionary
61             return data
```

The keys to this `data` dictionary are *not* arbitrary — these are the keys and values that TensorFlow *expects* an individual data point to have. A complete review of the TensorFlow dataset API is outside the scope of this book. If you're interested in learning more about this encoding, please refer to the TensorFlow documentation [45].

Line 61 returns the TensorFlow-compatible object to our calling function, enabling us to add the object to the record dataset.

16.3.4 Building the LISA + TensorFlow Dataset

In order to train a network using the TFOD API, we first need to convert our images and annotations into TensorFlow record format, similar to how we converted an image dataset to mxnet's format. Let's go ahead and convert our sample of the LISA dataset — open up the `build_lisa_records.py` file and insert the following code:

```
1  # import the necessary packages
2  from config import lisa_config as config
3  from pyimagesearch.utils.tfannotation import TFAnnotation
4  from sklearn.model_selection import train_test_split
5  from PIL import Image
6  import tensorflow as tf
7  import os
```

Lines 2-7 import our required Python packages. **Line 2** imports our special configuration file so we have access to our input image paths, annotations, file, and output record files. On **Line 3** we import our custom `TFAnnotation` class to make our actual `build_lisa_records.py` cleaner and easier to read.

Next, let's start defining the `main` method which will be executed when we run our script:

```
9  def main(_):
10     # open the classes output file
11     f = open(config.CLASSES_FILE, "w")
12
13     # loop over the classes
14     for (k, v) in config.CLASSES.items():
15         # construct the class information and write to file
16         item = ("item {\n"
17                 "\tid: " + str(v) + "\n"
18                 "\tname: '" + k + "'\n"
19                 "}\n")
20         f.write(item)
21
22     # close the output classes file
```

```
23      f.close()
24
25      # initialize a data dictionary used to map each image filename
26      # to all bounding boxes associated with the image, then load
27      # the contents of the annotations file
28      D = {}
29      rows = open(config.ANNOT_PATH).read().strip().split("\n")
```

On **Line 11** we open a file pointer to our output class labels file. **Lines 13-20** then loop over each of the class labels, writing each to file. The TFOD API expects this file in a certain a JSON/YAML-like format.

Each class must have an `item` object with two attributes: `id` and `name`. The `name` is the human-readable name for the class label while the `id` is an unique integer for the class label. Keep in mind that the class IDs should start counting from `1` rather than `0` as zero is reserved for the "background" class.

Line 28 initialize a data dictionary used to map each individual image filename to its respective bounding boxes and annotations. **Line 29** then loads the contents of our annotations file.

The format of this annotations file will become clear in the next code block:

```
31      # loop over the individual rows, skipping the header
32      for row in rows[1:]:
33          # break the row into components
34          row = row.split(",")[0].split(";")
35          (imagePath, label, startX, startY, endX, endY, _) = row
36          (startX, startY) = (float(startX), float(startY))
37          (endX, endY) = (float(endX), float(endY))
38
39          # if we are not interested in the label, ignore it
40          if label not in config.CLASSES:
41              continue
```

On **Line 32** we start looping over each of the individual rows in the annotations file, skipping the header. **Line 34** breaks the comma separated row into a list. **Line 35** then extracts the values from the rows, thus enabling us to see there are six important components to our CSV annotations file:

1. The input path to the image file
2. The class label
3. The starting x-coordinate for the bounding box
4. The starting y-coordinate for the bounding box
5. The ending x-coordinate for the bounding box
6. The ending y-coordinate for the bounding box

Lines 36 and 37 convert the bounding box coordinates from strings to floats. If the row we are currently examining is not included in our list of `CLASSES`, we ignore it (**Lines 40 and 41**).

Since an image can contain multiple traffic signs, and therefore multiple bounding boxes, we need to utilize a Python dictionary to *map* the image path (as the key) to a list of labels and associated bounding boxes (the value):

```
43          # build the path to the input image, then grab any other
44          # bounding boxes + labels associated with the image
45          # path, labels, and bounding box lists, respectively
```

```
46          p = os.path.sep.join([config.BASE_PATH, imagePath])
47          b = D.get(p, [])
48
49          # build a tuple consisting of the label and bounding box,
50          # then update the list and store it in the dictionary
51          b.append((label, (startX, startY, endX, endY)))
52          D[p] = b
```

Line 46 builds the full path to the current `imagePath` using our `BASE_PATH` from the config file. We then fetch all bounding boxes + class labels for the current image path, `p` (**Line 47**). We append our `label` and bounding box coordinates to the list `b`, and then store it back in the mapping dictionary, `D` (**Lines 51 and 52**).

This mapping may seem unimportant, but keep in mind that we still need to:

1. Create our training and testing split
2. Ensure that our training and testing split is performed on the *images* rather than the *bounding boxes*

Our goal here is to ensure that if a particular image is labeled as "training", then *all* bounding boxes for that image is included in the training set. Similarly, if an image is labeled as "testing", then we want *all* bounding boxes for that image to be associated with the testing set.

We want to *avoid* situations where an image contains bounding boxes for *both* training and testing sets. Not only is this behavior inefficient, it creates a larger problem — some object detection algorithms utilize hard-negative mining to increase their accuracy by taking non-labeled areas of the image and treating them as negatives.

It's therefore possible that our network could *think* a particular ROI was *not* a road sign, when in fact it was, thereby confusing our network and hurting accuracy. Because of this nuance, you should always associate the *image* and not a *bounding box* with either the training or testing set.

Speaking of which, let's create our training/testing split now:

```
54      # create training and testing splits from our data dictionary
55      (trainKeys, testKeys) = train_test_split(list(D.keys()),
56          test_size=config.TEST_SIZE, random_state=42)
57
58      # initialize the data split files
59      datasets = [
60          ("train", trainKeys, config.TRAIN_RECORD),
61          ("test", testKeys, config.TEST_RECORD)
62      ]
```

We use scikit-learn's `train_test_split` to create our split. Notice how we are splitting on the *keys* of the `D` mapping dictionary, ensuring we split on the image paths rather than the bounding boxes, avoiding the potentially harmful scenario mentioned above.

Lines 59-62 define a `datasets` list, associating the training and testing splits with their associated output files.

At this point we are ready to build our TensorFlow record files:

```
64      # loop over the datasets
65      for (dType, keys, outputPath) in datasets:
66          # initialize the TensorFlow writer and initialize the total
67          # number of examples written to file
68          print("[INFO] processing '{}'...".format(dType))
69          writer = tf.python_io.TFRecordWriter(outputPath)
```

```
70          total = 0
71
72          # loop over all the keys in the current set
73          for k in keys:
74               # load the input image from disk as a TensorFlow object
75               encoded = tf.gfile.GFile(k, "rb").read()
76               encoded = bytes(encoded)
77
78               # load the image from disk again, this time as a PIL
79               # object
80               pilImage = Image.open(k)
81               (w, h) = pilImage.size[:2]
```

We start looping over our training and testing splits on **Line 65**. We instantiate our `writer`, a `TFRecordWriter` which points to our current `outputPath` on **Line 69**.

Line 73 starts looping over the keys to our `D` mapping dictionary for the current split. Keep in mind that `k` is actually an image path — using this path we can load the encoded image in TensorFlow format on **Lines 75 and 76**.

It is possible to extract the width and height from a TensorFlow encoded object, but it's a bit tedious — I prefer to simply load the image in PIL/Pillow format and extract the dimensions (**Lines 80 and 81**). Loading the image a second time does require an additional I/O call, but this script is only executed once per dataset — I prefer clean, understandable, less-buggy code in this circumstance.

We can now build our `tfAnnot` object:

```
83               # parse the filename and encoding from the input path
84               filename = k.split(os.path.sep)[-1]
85               encoding = filename[filename.rfind(".") + 1:]
86
87               # initialize the annotation object used to store
88               # information regarding the bounding box + labels
89               tfAnnot = TFAnnotation()
90               tfAnnot.image = encoded
91               tfAnnot.encoding = encoding
92               tfAnnot.filename = filename
93               tfAnnot.width = w
94               tfAnnot.height = h
```

Lines 84 and 85 extract the `filename` and image `encoding` (i.e., JPG, PNG, etc.) from the file path.

Lines 89-94 instantiate the `tfAnnot` object and set the TensorFlow encoded image, file encoding, filename, and width and height. We haven't added any bounding box information to `tfAnnot`, so let's take care of that now:

```
96               # loop over the bounding boxes + labels associated with
97               # the image
98               for (label, (startX, startY, endX, endY)) in D[k]:
99                    # TensorFlow assumes all bounding boxes are in the
100                   # range [0, 1] so we need to scale them
101                   xMin = startX / w
102                   xMax = endX / w
103                   yMin = startY / h
```

```
104                     yMax = endY / h
105
106                     # update the bounding boxes + labels lists
107                     tfAnnot.xMins.append(xMin)
108                     tfAnnot.xMaxs.append(xMax)
109                     tfAnnot.yMins.append(yMin)
110                     tfAnnot.yMaxs.append(yMax)
111                     tfAnnot.textLabels.append(label.encode("utf8"))
112                     tfAnnot.classes.append(config.CLASSES[label])
113                     tfAnnot.difficult.append(0)
114
115                     # increment the total number of examples
116                     total += 1
```

Line 98 loops over all labels and bounding boxes associated with the image path, `k`.

TensorFlow assumes all bounding boxes are in the range $[0, 1]$ so we perform a scaling step on **Lines 101-104** by dividing the bounding box coordinates by their associated with or height.

Lines 107-113 add the class label and bounding box information to the `tfAnnot` object. We'll also increment our `total` count so we can keep track of the total number of bounding boxes for each training/testing split, respectively.

Our final code block handles encoding the data point attributes in TensorFlow format, adding the `example` to the writer, and ensuring that our `main` method is executed via TensorFlow when we start our `build_lisa_records.py` script:

```
118                 # encode the data point attributes using the TensorFlow
119                 # helper functions
120                 features = tf.train.Features(feature=tfAnnot.build())
121                 example = tf.train.Example(features=features)
122
123                 # add the example to the writer
124                 writer.write(example.SerializeToString())
125
126             # close the writer and print diagnostic information to the
127             # user
128             writer.close()
129             print("[INFO] {} examples saved for '{}'".format(total,
130                 dType))
131
132 # check to see if the main thread should be started
133 if __name__ == "__main__":
134     tf.app.run()
```

To build the LISA records file, open up a terminal, and execute the following command:

```
$ time python build_lisa_records.py
[INFO] processing 'train'...
[INFO] 2876 examples saved for 'train'
[INFO] processing 'test'...
[INFO] 955 examples saved for 'test'

real    0m4.879s
user    0m3.117s
sys     0m2.580s
```

Here you can see that the entire process took only four seconds. Examining your `lisa/records` directory you should see our training and testing files:

```
$ ls lisa/records/
classes.pbtxt  testing.record  training.record
```

As you can see, we have successfully created our record files and associated class labels file in TensorFlow format.

16.3.5 A Critical Pre-Training Step

At this point we can proceed to training our Faster R-CNN; however, I want to end with a word of warning:

One of the **biggest mistakes** I see deep learning developers, students, and researchers make is *rushing* and not double-checking their work when building a dataset.

Whether your school project is due tonight at midnight, your boss is breathing down your neck for the latest and greatest model, or you're simply tinkering with deep learning as a hobby, be warned: rushing will only cause you problems, *especially* in the context of object detection.

Keep in mind that we are working with *more* than just an input image and a class label — we now have four additional components: **the bounding box coordinates themselves**. Too often I see people *assume* their code is correct, and if the script executes and completes without error, then everything must be okay. **Don't fall into this trap.**

Before you even *think* about training your network, double-check your code. In particular, go through **Lines 98-113** when we build the bounding box and class label for a given image. You should *visually validate* your code is working as you think it should by:

1. Loading the image from disk via OpenCV
2. Drawing the bounding box coordinates on the image by scaling `xMin`, `xMax`, `yMin`, and `yMax` back to standard integer coordinates
3. Drawing the class label on the image as well

Below I have included a code segment on how I perform this process:

```
96                # loop over the bounding boxes + labels associated with
97                # the image
98                for (label, (startX, startY, endX, endY)) in D[k]:
99                     # TensorFlow assumes all bounding boxes are in the
100                    # range [0, 1] so we need to scale them
101                    xMin = startX / w
102                    xMax = endX / w
103                    yMin = startY / h
104                    yMax = endY / h
105
106                    # load the input image from disk and denormalize the
107                    # bounding box coordinates
108                    image = cv2.imread(k)
109                    startX = int(xMin * w)
110                    startY = int(yMin * h)
111                    endX = int(xMax * w)
112                    endY = int(yMax * h)
113
114                    # draw the bounding box on the image
115                    cv2.rectangle(image, (startX, startY), (endX, endY),
116                         (0, 255, 0), 2)
```

```
117
118                     # show the output image
119                     cv2.imshow("Image", image)
120                     cv2.waitKey(0)
```

I won't inspect every single image in my dataset, but I'll sample a few hundred for each class and spend 10-20 minutes looking at them for errors. This visual validation is ***critical*** and ***should not be skipped under any circumstance***.

Do not fall into the false assumption that just because your code executes and completes that everything is working fine. Instead, take a pessimistic approach: ***nothing*** **is working fine until you take the time to double-check your work.**

If you wish to work with your own custom datasets and build the associated record files, you should use this script as a starting point and modify the class label/bounding box parsing. Try not to modify the actual construction of the `TFAnnotation` object as best as you can, ensuring you minimize the potential of inserting bugs.

While you can certainly parse out information from the TensorFlow record files after they are built, it's a tedious, time consuming process and your time and effort is better spent before the annotation object is added to the record.

Again, I cannot stress the importance of checking your bounding boxes enough — take your time, avoid costly errors, both in terms of time and finances spent training your network.

If you are not obtaining satisfiable accuracy on a particular object detection dataset, none of my hyperparameter tuning recommendations matter or will have an impact if you do not take the time to validate your dataset before you move on to training — please, for your sake, keep this in mind when working with your own data.

16.3.6 Configuring the Faster R-CNN

Training our Faster R-CNN on the LISA dataset is a four step process:

1. Download the pre-trained Faster R-CNN so we can fine-tune the network
2. Download the sample TFOD API configuration file and modify it to point to our record files
3. Start the training process and monitor
4. Export the frozen model graph after training is complete

Links can and will change over time. I will do my absolute best to keep this book up to date, but if you find a link is 404'ing or the resulting page does not look like a screenshot included in this book, please refer to the companion website (http://pyimg.co/fnkxk) for the most up-to-date links.

To download our pre-trained model, you'll first want to head to the TensorFlow Object Detection Model Zoo: http://pyimg.co/1z34r

Here you'll find a table that lists various models that have been trained on the Common Objects in Context (COCO) dataset [59].

The COCO dataset contains over 200,000 labeled images. Networks trained on the COCO challenge can detect 20 class labels, including airplanes, bicycles, cars, dogs, cats, and more. Networks trained on the large COCO dataset typically can be fine-tuned to recognize objects in other datasets as well.

To download the model, find the Faster R-CNN + ResNet-101 + COCO link in the table (Figure 16.2) and download it. At the time of this writing, the filename is `faster_rcnn_resnet101_coco_2018_01_28.tar.gz`. Again, make sure you check the table on the TFOD API GitHub page to download the latest model). For additional help downloading the pre-trained model and configuring your development environment, please refer to Section 16.2 above.

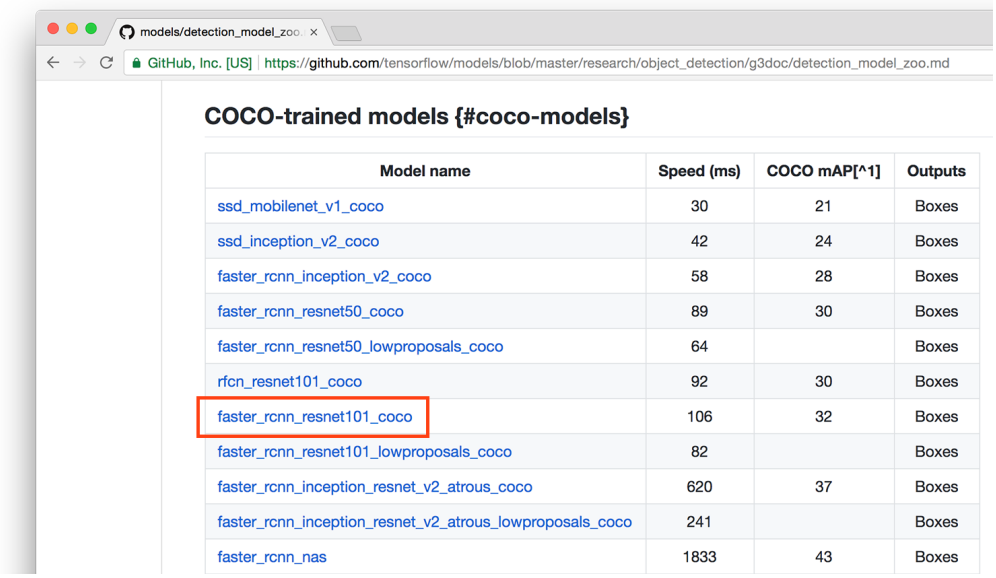


Figure 16.2: For this example, we will be using the Faster R-CNN + ResNet-101 architecture, trained on the COCO dataset. We will be fine-tuning this model for traffic sign detection.

After you have downloaded the model, move it to the `experiments/training` subdirectory and untar it:

```
$ cd lisa/experiments/training
$ mv ~/Downloads/faster_rcnn_resnet101_coco_2018_01_28.tar.gz ./
$ tar -zxvf faster_rcnn_resnet101_coco_2018_01_28.tar.gz
$ ls -l faster_rcnn_resnet101_coco_2018_01_28
checkpoint
frozen_inference_graph.pb
model.ckpt.data-00000-of-00001
model.ckpt.index
model.ckpt.meta
pipeline.config
saved_model
```

The files inside `faster_rcnn_resnet101_coco_2018_01_28` constitute the pre-trained TensorFlow Faster R-CNN on the COCO dataset. The exact structure of of this directory will become clear as we work through the chapter — we will defer an explanation of each file until the appropriate sections, respectively.

Now that we have our model weights, we also need a configuration file to instruct the TFOD API on how to train/fine-tune our network. Trying to define your own TFOD API configuration file from scratch would be an extremely trying, tedious, and arduous process. Instead, it's recommended to take one of the existing *configuration* files and update the paths and any other settings you see fit.

The full list of sample TFOD API configuration files can be found on this page: http://pyimg.co/r2xql. You'll see there are four files for the Faster R-CNN + ResNet-101 architecture (Figure 16.3, outlined in red) — make sure you download the `faster_rcnn_resnet101_pets.config` and save it to the `experiments/training` directory:

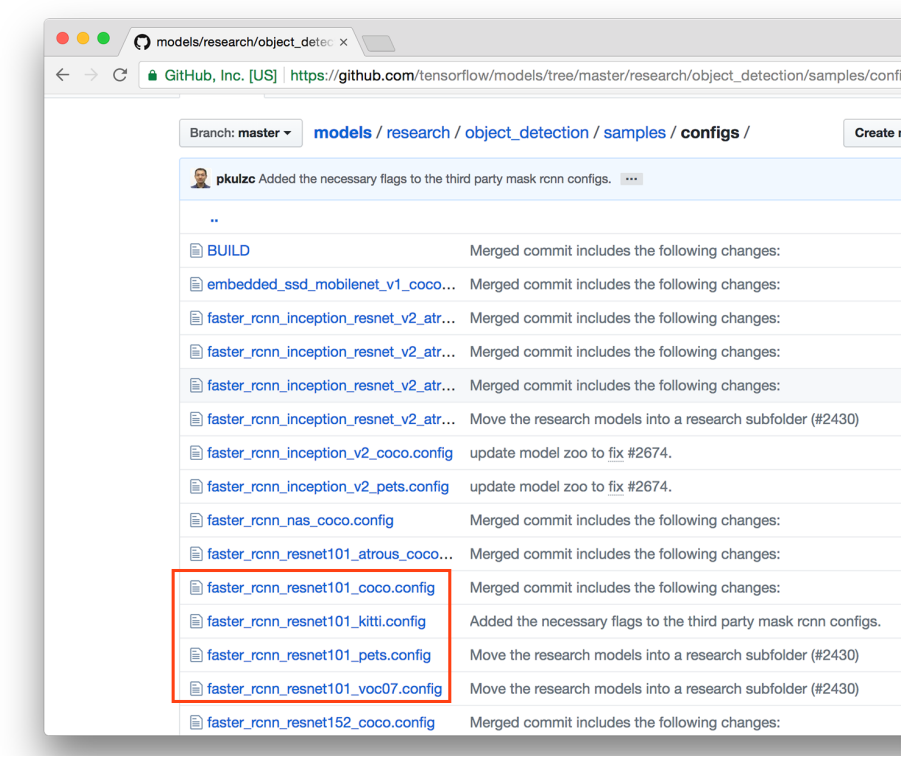


Figure 16.3: Along with our frozen model weights, we also need a starter configuration file. The TFOD API provides a number of configuration files for the Faster R-CNN + ResNet-101 architecture.

```
$ cd lisa/experiments/training
$ mv ~/Downloads/faster_rcnn_resnet101_pets.config faster_rcnn_lisa.config
```

I have also taken the extra step of renaming the configuration file to `faster_rcnn_lisa.config`.

You may be confused why we are using the configuration associated with the Oxford-IIIT "Pets" Dataset [60] rather than the COCO dataset the network was originally trained on. The short answer is that the Pets configuration requires fewer changes than the COCO configuration (at least, at the time of this writing) — the Pets configuration tends to be a better starting point.

Opening up our `faster_rcnn_lisa.config` configuration you'll see a large number of attributes in a JSON/YAML-like configuration:

```
$ vi faster_rcnn_lisa.config
model {
  faster_rcnn {
    num_classes: 37
    image_resizer {
      keep_aspect_ratio_resizer {
        min_dimension: 600
        max_dimension: 1024
      }
    }
    feature_extractor {
      type: 'faster_rcnn_resnet101'
```

```
        first_stage_features_stride: 16
      }
...
```

Luckily, there are only seven configurations we need to update, five of which are file paths that can be easily copied and pasted.

Let's start by setting the number of `num_classes` — this value is originally `37` but we'll change it to `3`, the total number of classes (pedestrian crossing, signal ahead, and stop sign) in our LISA dataset sample:

```
8   faster_rcnn {
9     num_classes: 3
10    image_resizer {
11      keep_aspect_ratio_resizer {
12        min_dimension: 600
13        max_dimension: 1024
14      }
15    }
```

You'll then want to update the `num_steps`, which is the number of `batch_size` steps to perform when training:

```
84  train_config: {
85    batch_size: 1
86  ...
87    num_steps: 50000
88    data_augmentation_options {
89      random_horizontal_flip {
90      }
91    }
92  }
```

This value defaults at 200,000, but I recommend using 50,000 for this particular dataset. For *most* datasets you'll want to train a bare minimum of 20,000 steps. You can certainly train for more steps, *especially* if you see your network performance improving with time, but I would typically not even bother stopping training until after 20,000 steps.

You will typically see a `batch_size=1` for Faster R-CNNs using the TFOD API. You can adjust this value as you see fit in your own experiments, provided your GPU has enough memory.

You'll notice that the configuration API conveniently has the text `PATH_TO_BE_CONFIGURED` placed throughout the configuration file wherever you need to update a file path. The first one is the `fine_tune_checkpoint`, which is the path to our downloaded Faster R-CNN + ResNet 101 `model.ckpt` base file:

```
109   gradient_clipping_by_norm: 10.0
110   fine_tune_checkpoint: "/home/adrian/pyimagesearch/dlbook/ssds_and_rcnn/lisa/
111        experiments/training/faster_rcnn_resnet101_coco_2018_01_28/model.ckpt"
112   from_detection_checkpoint: true
```

Make sure you supply the *full path* to this file *without* any console shortcuts, such as the `~` directory operator.

If you examine the contents of `faster_rcnn_resnet101_coco_2018_01_28` you'll notice there is no actual `model.cpkt` file but there are three files that *start* with `model.cpkt` — this is expected. The `model.cpkt` file is the *base filename* in which the TFOD API uses to derive the other three files.

The next two `PATH_TO_BE_CONFIGURED` updates can be found inside the `train_input_reader`, in particular the `input_path` and `label_map_path`:

```
train_input_reader: {
  tf_record_input_reader {
    input_path: "/home/adrian/pyimagesearch/dlbook/ssds_and_rcnn/lisa/
        records/training.record"
  }
  label_map_path: "/home/adrian/pyimagesearch/dlbook/ssds_and_rcnn/lisa/
      records/classes.pbtxt"
}
```

You'll want to update these paths to point to your `training.record` file and `classes.pbtxt` file, respectively. Again, it's likely that your directory structure will be slightly different than mine so *do not* copy and paste my paths — make sure you double-check your paths and include them in the configuration file.

The `eval_input_reader` requires us to update the `input_path` and `label_map_path` as well:

```
eval_input_reader: {
  tf_record_input_reader {
    input_path: "/home/adrian/pyimagesearch/dlbook/ssds_and_rcnn/lisa/
        records/testing.record"
  }
  label_map_path: "/home/adrian/pyimagesearch/dlbook/ssds_and_rcnn/lisa/
      records/classes.pbtxt"
  shuffle: false
  num_readers: 1
}
```

But this time make sure you include the path to the `testing.record` file rather than the `training.record` file.

I would also suggest updating your `eval_config` as well:

```
eval_config: {
  num_examples: 955
  # Note: The below line limits the evaluation process to 10 evaluations.
  # Remove the below line to evaluate indefinitely.
  #max_evals: 10
}
```

Set the `num_examples` to be the total number of bounding boxes in the testing set. I also like to comment out the `max_evals` as this will ensure the evaluation script runs indefinitely until we manually stop it, but that decision is up to you and your hardware. We will discuss how you should properly set `max_evals` and run the evaluation script in Section 16.3.8.

16.3.7 Training the Faster R-CNN

We are now ready to train our Faster R-CNN + ResNet-101 architecture on the LISA Traffic Signs dataset! It may seem like it has taken a lot of steps to get to this point, but keep in mind two key aspects:

1. The TFOD API is meant to be extendible, reusable, and configurable so by definition there will be more configuration files. Using these configuration files is actually a benefit to you, the end user, as it enables you to write less code and focus on the actual model parameters.
2. The process becomes significantly easier the more and more you run these experiments.

Since we are using the TFOD API, we do not have to write any actual code to launch the training process — the only custom code required on our part is to build the record dataset, as we have done in the previous sections. From there, the scripts provided in the `models/research/object_detection` directory of the TFOD API can be used to train and evaluate our networks.

Training can be started using the following command (make sure you `source` the `setup.sh` script to export your `PYTHONPATH` before you run this command):

```
$ python object_detection/train.py --logtostderr \
     --pipeline ssds_and_rcnn/lisa/experiments/training/faster_rcnn_lisa.config \
     --train_dir ssds_and_rcnn/lisa/experiments/training
INFO:tensorflow:Starting Session.
INFO:tensorflow:Starting Queues.
INFO:tensorflow:global_step/sec: 0
INFO:tensorflow:Recording summary at step 0.
INFO:tensorflow:global step 1: loss = 2.3321 (5.715 sec/step)
INFO:tensorflow:global step 2: loss = 2.1641 (0.460 sec/step)
...
```

Notice how the `--pipeline` switch points to our `faster_rcnn_lisa.config` file. The `--train-dir` switch then points to our `training` subdirectory inside `data`.

The paths to the input files and directories can be quite long, so for convenience, I have created a sym-link from the `ssds_and_rcnn` directory to the `models/research` directory, enabling me to spend less time typing and debugging long file paths. You may wish to apply this tactic as well.

While the command above will start *training* the network, we need a second command to start evaluation. Open up a second terminal and issue the following command:

```
$ python object_detection/eval.py --logtostderr \
 --pipeline_config_path ssds_and_rcnn/lisa/experiments/training/faster_rcnn_lisa.config \
 --checkpoint_dir ssds_and_rcnn/lisa/experiments/training \
 --eval_dir ssds_and_rcnn/lisa/experiments/evaluation
...
```

Here we supply:

- `--pipeline_config_path` which points to our `faster_rcnn_lisa.config` file
- `--checkpoint_dir` which is the path to the `training` subdirectory and where all model checkpoints will be saved during training
- `--eval_dir`, which is the path to the `evaluation` subdirectory where the evaluation logs will be stored

Notice how the two above commands were executed inside the `models/research` directory of the TFOD API (I.e., where the `object_detection/train.py` and `object_detection/eval.py` scripts live).

The last command can be executed anywhere on your system, provided you supply the full path to the `data` directory containing our `training` and `evaluation` subdirectories:

```
$ cd ~/pyimagesearch/dlbook/ssds_and_rcnn/
$ tensorboard --logdir lisa/experiments
TensorBoard 0.1.8 at http://annalee:6006 (Press CTRL+C to quit)
```

The `tensorboard` command can be used to produce a nice visualization of the training process inside our web browser (i.e., no parsing logs like in mxnet). On my machine, I can visualize the training process by opening up a web browser and pointing it to `http://annalee:6006`.

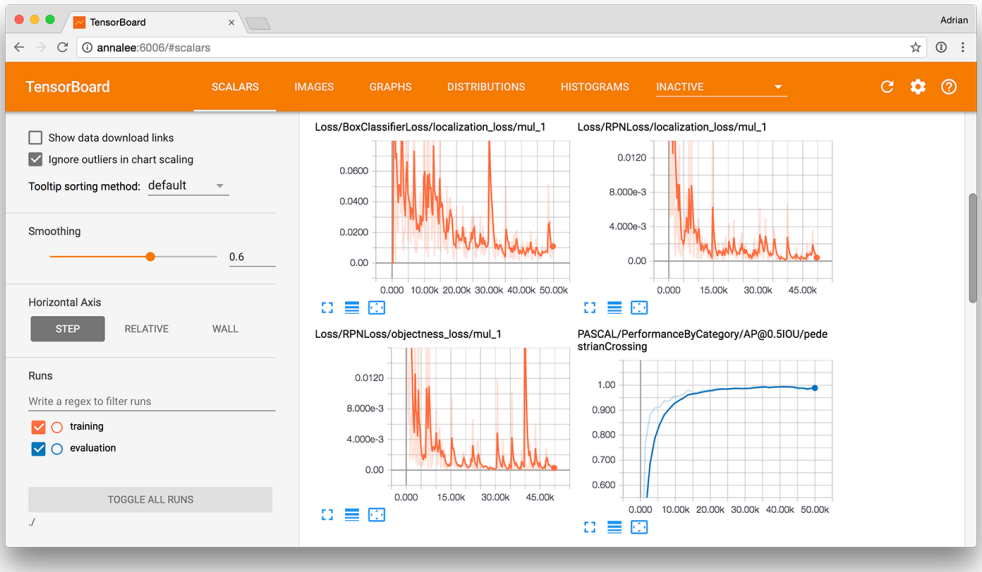


Figure 16.4: An example screenshot of my TensorBoard dashboard after 24 hours of training.

On your machine the port should still be `6006` but the hostname (i.e., `annalee`) will be either the IP address of your machine or the name of the machine itself.

Additionally, it may take a few minutes for the TensorBoard dashboard to populate — both `train.py` and `eval.py` need to run for awhile in order to generate the logs for TensorBoard to parse and display. Furthermore, some TensorBoard values, such as mAP, be be invisible until the `eval.py` script has a chance to run. Depending on the speed of your machine and evaluation process, it may take upwards of 15-30 minutes for the mAP values to display.

Figure 16.4 contains a screenshot of my TensorBoard dashboard. Here you can see loss steadily decreasing while my mAP grows, increasing as our network is learning.

Training will continue until either `num_steps` have been met or you `ctrl + c` out of the script — you'll also need to `ctrl + c` out of the `eval.py` and `tensorboard` as well.

By the time we reach step 50,000, my loss value is a ≈ 0.008 with a **mAP of 98%**, which is very good.

I'll wrap up this section by saying if this is your first time trying to train a network using the TFOD API, *you'll very likely run into errors.* Be sure to read Section 16.3.8 below as I have

listed suggestions for training your networks to help you (1) avoid errors entirely and (2) resolve them when errors appear.

16.3.8 Suggestions When Working with the TFOD API

This section includes a number of recommendations and best practices I personally utilize whenever working with the TFOD API. Take time to carefully read this section as it will help you diagnose errors and better train your own object detection networks.

Accept there is a Learning Curve

The first fact you need to accept is that there *is a learning curve* when using the TFOD API — and there is no way around it. When you first utilize the TFOD API, *do not* expect all scripts to work out-of-the-box.

To be frank, it's unrealistic. My suggestion would be to set aside ***at least* one to two days** (depending on your experience level with Python, Unix systems, and diagnosing errors) to familiarize yourself with the API.

If you expect your entire pipeline to be up and running in under an hour, especially if this is the first time you've used the TFOD API, your expectations are not aligned with reality — that's not how these state-of-the-art deep learning tools work.

Don't become frustrated. Don't become annoyed. And despite any confusion or frustration, don't leave your workstation and immediately run to your colleague, coworker, or fellow student for help. Errors with these tools will happen. It is expected.

Start by taking a deep breath and then take the time to diagnose the error, and again, accept there is a learning curve — the stack traces produced by TensorFlow are typically helpful and include what the error is (although you may need to scroll through the output to find exactly what the error is as the output can be a bit verbose).

In many situations the errors you run into will likely be due to not properly updating your Python path or your file paths. Always double-check your file paths!

Replicate Known Results First

My second recommendation is to train your first Faster R-CNN on the LISA Traffic Sign dataset as we do in this chapter. Whether or not you train the network over 50,000 steps is up to you, but I would train for *at least* 2,500 steps so you can validate that your training and evaluation pipeline is properly working — accuracy will still be low at this point but you'll at least be able to determine if the network is actually learning.

Too often I see deep learning students, engineers, researchers, and practitioners try to *immediately:*

1. Swap out the example datasets
2. Plug in their own data
3. Attempt to train the network
4. And become confused when there are errors or poor results

Don't do this. There are too many moving parts to immediately swap in your own dataset. You need to learn how to use the TFOD API tools *before* you can train a network on your own dataset.

Regardless if you are facing a project deadline, trying to complete your graduation/capstone requirement, or simply toying with a hobby project, the simple fact is this:

There is *no excuse* for not *first* replicating the example results discussed here *before* you work with your own data. The rationale behind this thought goes back to the scientific method — *don't change too many variables at one time.*

Right now your goal is to get to the top of the TFOD API learning curve. The best way to get to the top of the learning curve is to work with a dataset and example where you can clearly see the expected results — replicate these results before you move on to other experiments.

Export Your PYTHONPATH

Another common error I see is failing to update your `PYTHONPATH` when trying to execute either (1) the scripts inside the `models/research` directory of the TFOD API or (2) trying to import files from the TFOD API. If you forget to update your `PYTHONPATH` your error will likely look similar to this:

```
$ python build_lisa_records.py
Traceback (most recent call last):
  File "build_lisa_records.py", line 6, in <module>
    from pyimagesearch.utils.tfannotation import TFAnnotation
  File "/home/adrian/.virtualenvs/dl4cv/lib/python3.4/site-packages/
  pyimagesearch/utils/tfannotation.py", line 2, in <module>
    from object_detection.utils.dataset_util import bytes_list_feature
ImportError: No module named 'object_detection'
```

Here we are trying to execute `build_lisa_records.py` which is trying to import functions from the `object_detection` sub-module of the TFOD API; however, since our `PYTHONPATH` is not properly set, the import fails.

In this case, export your `PYTHONPATH` using either the `setup.sh` script or manually via your shell:

```
$ source setup.sh
$ python build_lisa_records.py
[INFO] processing '{train}'...
[INFO] processing '{test}'...
```

From there Python will be able to find the TFOD API libraries.

Cleanup Your "training" Directory

When training my own networks I encountered a strange bug where my network training would *dramatically* slow down if I didn't occasionally delete all files in my `training` directory besides my `faster_rcnn_resnet101_coco_2018_01_28.tar.gz` and `faster_rcnn_lisa.config` files.

I'm not entirely sure why this is, but if you find your networks are training slower or your results don't seem right, stop training and then clean up the `training` directory:

```
$ cd lisa/experiments/training/
$ rm checkpoint graph.pbtxt pipeline.config
$ rm events.* model.*
$ rm -rf faster_rcnn_resnet101_coco_2018_01_28
$ tar -zxvf faster_rcnn_resnet101_coco_2018_01_28.tar.gz
$ ls -l
faster_rcnn_lisa.config
faster_rcnn_resnet101_coco_2018_01_28
faster_rcnn_resnet101_coco_2018_01_28.tar.gz
```

Here you can see I have deleted all checkpoints (including the original untarred Faster R-CNN model pre-trained on COCO) and event files. From there, restart training. Again, this bug/behavior may be specific to my machine or my particular setup, but I wanted to share it with you as well.

Monitor Loss and mAP

When training your own object detectors be sure to measure your loss and mAP values (the mAP will only be computed if you are running the `eval.py` script).

The mAP should start off low, rise quickly over the first 10,000-50,000 steps (depending on how long you are training for), and then level off. Your loss should decrease during this time as well. Your goal here is to drive the loss as low as you can, ideally < 1.5. A loss < 1 is desirable when training your own object detectors.

Training and Evaluation with GPUs and CPUs

Training a network with the TFOD API can be done using multiple GPUs, but, at the time of this writing, it's advisable to use a single GPU until you are familiar with the TFOD API tools. When running the evaluation script you should use a single GPU (if available) as well.

If you have only a single GPU on your machine you *should not* try to train and evaluate on the same GPU — you will likely run into a memory exhausted error and your scripts will exit.

Instead, you should kickoff the training process on your GPU and then run evaluation on your CPU. The catch here is that evaluating with your CPU will be incredibly slow; therefore, you should set `num_examples` to a very small number (such as 5-10) inside the `eval_config` of your pipeline configuration:

```
130   eval_config: {
131   num_examples: 5
132   # Note: The below line limits the evaluation process to 10 evaluations.
133   # Remove the below line to evaluate indefinitely.
134   #max_evals: 10
135 }
```

This update will ensure that a small fraction of your evaluation dataset is used for evaluation (allowing you to compute the approximate mAP) but not the *entire* testing set (which would take a long time on a single CPU).

Try to keep your CPU load as low as possible here so you can use those cycles for moving data on and off the GPU, thereby reducing the amount of time it takes to train the network.

Forgetting to Set your CUDA Visible Devices

Going hand-in-hand with Section 16.3.8 above, I've found that some readers are *aware* that they need to use a GPU for training and a CPU (if no additional GPUs are available) for evaluation, but are not sure how to accomplish this.

Reading the TensorFlow documentation (or most any documentation for deep learning libraries) you'll come across the `CUDA_VISIBLE_DEVICES` environment variable. By default, TensorFlow will allocate your model to GPUs on your machine when running a script, but somewhat confusingly to readers new to TensorFlow, the library will *not* train the network on all GPUs — it will only allocate *memory* on each of the GPUs. This behavior is problematic if you want a script to run on a *single* GPU or *no* GPU at all.

On my machine I have four Titan X GPUs which, by default, TensorFlow will try to allocate memory on all four of these GPUs. To prevent this undesired allocation from happening I can set my `CUDA_VISIBLE_DEVICES` before executing the script:

```
$ export CUDA_VISIBLE_DEVICES="0"
$ python object_detection/train.py --logtostderr \
    --pipeline ssds_and_rcnn/lisa/experiments/training/faster_rcnn_lisa.config \
    --train_dir ssds_and_rcnn/lisa/experiments/training
...
```

Here you can see that I am instructing my current shell session to *only* expose GPU `0` to TensorFlow. From there, only GPU `0` will be allocated when running `train.py`.

I can then open up a second terminal and set `CUDA_VISIBLE_DEVICES` to expose only GPU `1`:

```
$ export CUDA_VISIBLE_DEVICES="1"
$ python object_detection/eval.py --logtostderr \
 --pipeline_config_path ssds_and_rcnn/lisa/experiments/training/faster_rcnn_lisa.config \
 --checkpoint_dir ssds_and_rcnn/lisa/experiments/training \
 --eval_dir ssds_and_rcnn/lisa/experiments/evaluation
...
```

Here only GPU `1` is visible to TensorFlow and therefore TensorFlow will *only* use GPU `1` for eval.py — in this manner I can allocate specific GPUs to specific scripts.

Of course, if you *do not* want TensorFlow to use a GPU for a script, typical behavior when your machine only has one GPU and this GPU is utilized for training, you can set `CUDA_VISIBLE_DEVICES` to be empty:

```
$ export CUDA_VISIBLE_DEVICES=""
$ python object_detection/eval.py --logtostderr \
 --pipeline_config_path ssds_and_rcnn/lisa/experiments/training/faster_rcnn_lisa.config \
 --checkpoint_dir ssds_and_rcnn/lisa/experiments/training \
 --eval_dir ssds_and_rcnn/lisa/experiments/evaluation
...
```

The above command will instruct TensorFlow there are no GPUs (implied by the blank string in the export command) and that the CPU should be used instead when executing `eval.py`.

Optimizer Parameters

Looking at our pipeline configuration you'll see that we haven't fiddled with any of the optimizer parameters. In the case of Faster R-CNN + ResNet-101, the default configuration file uses a SGD + Nesterov momentum optimizer with a learning rate of 0.0003 — **you should keep these optimizer parameters as-is until you are comfortable with the toolkit and ideally have replicated the results of this chapter a handful of times.**

Additionally, be sure to read through the comments in the pipeline configuration from the TensorFlow community as they give you suggested values for parameters as well. A full review of all possible optimizer parameters inside the TFOD API is well outside the scope of this book, but provided you have read through the *Starter Bundle* and *Practitioner Bundle*, you'll find the parameters used inside the TFOD API closely match Keras and mxnet. Be sure to refer to the TFOD API documentation for questions regarding these parameters: http://pyimg.co/pml2d [45].

Respect the Toolkit

After going through this section you'll see there are a number of recommendations and best practices I suggest, but none of them is as important as Section 16.3.8 above — **there is a learning curve and you need to dedicate the time and energy to climb it.**

The TFOD API is an *extremely* powerful tool and once you get used to it, you'll be able to quickly and easily train your own deep learning-based object detectors. But to get to that point, you'll need to be patient.

Inevitably, you'll run into an error. Take the time to read through it, digest it, and even spend some time Googling the error and reading the TensorFlow documentation. Do not expect a "quick fix", that expectation is typically unrealistic.

That said, more times than not, you'll find your error is due to either:

1. Forgetting to export your `PYTHONPATH`
2. A typo in a file path in a configuration file
3. A problem generating your record files, such as invalid bounding boxes

Respect the toolkit, invest your time in climbing the learning curve, and the toolkit will respect you.

16.3.9 Exporting the Frozen Model Graph

Now that our model is trained, we can use the `export_inference_graph.py` to create a TensorFlow model we can import into our own scripts. To export your model, first change directory to your `models/research` directory on your own machine and then execute `export_inference_graph.py`:

```
$ cd ~/models/research
$ python object_detection/export_inference_graph.py \
 --input_type image_tensor \
 --pipeline_config_path ~ssds_and_rcnn/lisa/experiments/training/faster_rcnn_lisa.config \
 --trained_checkpoint_prefix ~ssds_and_rcnn/lisa/experiments/training/model.ckpt-50000 \
 --output ~/ssds_and_rcnn/lisa/experiments/exported_model
```

Here you can see that I have exported the model checkpoint at step 50,000 to the `exported_model` directory. Checking the contents of `exported_model` you can see the files TensorFlow generated:

```
$ ls lisa/experiments/exported_model/
checkpoint  frozen_inference_graph.pb  model.ckpt.data-00000-of-00001
model.ckpt.index  model.ckpt.meta  saved_model
```

Be sure to keep your `exported_model` directory and the files inside it — we'll need to import these files into our Python script in the next section.

16.3.10 Faster R-CNN on Images and Videos

We've trained our Faster R-CNN. We've evaluated its accuracy. But we have yet to programmatically apply it to an input image — how do we go about applying our network to an input image *outside* the dataset it was trained on? To answer this question, open up `predict.py` and insert the following code:

```
# import the necessary packages
from object_detection.utils import label_map_util
import tensorflow as tf
import numpy as np
import argparse
import imutils
import cv2

# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-m", "--model", required=True,
     help="base path for frozen checkpoint detection graph")
ap.add_argument("-l", "--labels", required=True,
     help="labels file")
ap.add_argument("-i", "--image", required=True,
     help="path to input image")
```

```
17 ap.add_argument("-n", "--num-classes", type=int, required=True,
18     help="# of class labels")
19 ap.add_argument("-c", "--min-confidence", type=float, default=0.5,
20     help="minimum probability used to filter weak detections")
21 args = vars(ap.parse_args())
```

Lines 2-8 import our Python libraries. TensorFlow's `label_map_util` is a helper function that will enable us to easily load our class labels file from disk.

We then parse our command line arguments on **Lines 10-21**. We require four command line arguments, followed by one optional one:

1. `--model`: Here we specify the path to our frozen checkpoint detection graph (i.e., the TensorFlow model itself).
2. `--labels`: The labels are not built into the TensorFlow model, so we'll need to supply the path to our `classes.pbtxt` file.
3. `--image`: This switch controls the path to our input image that we want to apply object detection to.
4. `--num-classes`: Unfortunately, TensorFlow cannot automatically determine the number of classes from the `--labels` file so we need to manually specify this number as well.
5. `--min-confidence` (optional): The minimum probability used to filter out weak detections.

Now that our command line arguments are parsed, we can load the model from disk:

```
23 # initialize a set of colors for our class labels
24 COLORS = np.random.uniform(0, 255, size=(args["num_classes"], 3))
25
26 # initialize the model
27 model = tf.Graph()
28
29 # create a context manager that makes this model the default one for
30 # execution
31 with model.as_default():
32     # initialize the graph definition
33     graphDef = tf.GraphDef()
34
35     # load the graph from disk
36     with tf.gfile.GFile(args["model"], "rb") as f:
37         serializedGraph = f.read()
38         graphDef.ParseFromString(serializedGraph)
39         tf.import_graph_def(graphDef, name="")
```

Line 24 randomly initializes a set of RGB colors for each bounding box. The random initialization is done out of convenience — you can modify this script to use fixed colors per label as well.

Line 27 initializes the `model` that we'll be loading from disk. **Lines 31-39** then load the serialized TensorFlow network using TensorFlow's helper utilities.

Let's load our class labels from disk as well:

```
41 # load the class labels from disk
42 labelMap = label_map_util.load_labelmap(args["labels"])
43 categories = label_map_util.convert_label_map_to_categories(
44     labelMap, max_num_classes=args["num_classes"],
45     use_display_name=True)
46 categoryIdx = label_map_util.create_category_index(categories)
```

Line 42 loads the raw `pbtxt` file from disk. We can then use the `convert_label_map_to_categories` function along with our `--numclasses` switch to build the set of `categories`. **Line 46** creates a mapping from the *integer ID* of the class label (i.e., what TensorFlow will return when predicting) to the human readable class label.

In order to predict bounding boxes for our input image we first need to create a TensorFlow session and grab references to each of the image, bounding box, probability, and classes tensors inside the network:

```
48 # create a session to perform inference
49 with model.as_default():
50     with tf.Session(graph=model) as sess:
51         # grab a reference to the input image tensor and the boxes
52         # tensor
53         imageTensor = model.get_tensor_by_name("image_tensor:0")
54         boxesTensor = model.get_tensor_by_name("detection_boxes:0")
55
56         # for each bounding box we would like to know the score
57         # (i.e., probability) and class label
58         scoresTensor = model.get_tensor_by_name("detection_scores:0")
59         classesTensor = model.get_tensor_by_name("detection_classes:0")
60         numDetections = model.get_tensor_by_name("num_detections:0")
```

These references will enable us to access their associated values after passing the image through the network.

Next, let's load our image from disk and prepare it for detection:

```
62         # load the image from disk
63         image = cv2.imread(args["image"])
64         (H, W) = image.shape[:2]
65
66         # check to see if we should resize along the width
67         if W > H and W > 1000:
68             image = imutils.resize(image, width=1000)
69
70         # otherwise, check to see if we should resize along the
71         # height
72         elif H > W and H > 1000:
73             image = imutils.resize(image, height=1000)
74
75         # prepare the image for detection
76         (H, W) = image.shape[:2]
77         output = image.copy()
78         image = cv2.cvtColor(image.copy(), cv2.COLOR_BGR2RGB)
79         image = np.expand_dims(image, axis=0)
```

This code block should feel similar to what we have previous done with Keras and mxnet. We start by loading the image from disk, grabbing its dimensions, and resizing it such that the largest dimension is, at most, 1,000 pixels (**Lines 63-73**). You can resize the image either larger or smaller as you see fit, but 300-1,000 pixels is a good starting point. **Lines 75-79** then prepare the image for detection.

Obtaining the bounding boxes, probabilities, and class labels is as simple as calling the `.run` method of the session:

```
81          # perform inference and compute the bounding boxes,
82          # probabilities, and class labels
83          (boxes, scores, labels, N) = sess.run(
84               [boxesTensor, scoresTensor, classesTensor, numDetections],
85               feed_dict={imageTensor: image})
86
87          # squeeze the lists into a single dimension
88          boxes = np.squeeze(boxes)
89          scores = np.squeeze(scores)
90          labels = np.squeeze(labels)
```

Here we pass our list of bounding box, scores (i.e., probabilities), class labels, and number of detections tensors into the `sess.run` method. The `feed_dict` instructs TensorFlow to set the `imageTensor` to our `image` and run a forward-pass, yielding our bounding boxes, scores (i.e., probabilities), and class labels.

The `boxes`, `scores`, and `labels` are all multi-dimensional arrays so we squeeze them to a 1D array, enabling us to easily loop over them.

If you are unfamiliar with NumPy's squeeze method and how it works, be sure to refer to the documentation: http://pyimg.co/fnbgd

Figure 16.5: Examples of applying our Faster R-CNN to detect and label the presence of US traffic signs in images.

The final step is to loop over our detections, draw them on the `output` image, and display the result to our screen:

```
92          # loop over the bounding box predictions
93          for (box, score, label) in zip(boxes, scores, labels):
94               # if the predicted probability is less than the minimum
95               # confidence, ignore it
96               if score < args["min_confidence"]:
97                    continue
98
99               # scale the bounding box from the range [0, 1] to [W, H]
100              (startY, startX, endY, endX) = box
101              startX = int(startX * W)
102              startY = int(startY * H)
103              endX = int(endX * W)
104              endY = int(endY * H)
105
106              # draw the prediction on the output image
107              label = categoryIdx[label]
108              idx = int(label["id"]) - 1
109              label = "{}: {:.2f}".format(label["name"], score)
110              cv2.rectangle(output, (startX, startY), (endX, endY),
111                   COLORS[idx], 2)
112              y = startY - 10 if startY - 10 > 10 else startY + 10
113              cv2.putText(output, label, (startX, y),
114                   cv2.FONT_HERSHEY_SIMPLEX, 0.3, COLORS[idx], 1)
115
116         # show the output image
117         cv2.imshow("Output", output)
118         cv2.waitKey(0)
```

On **Line 93** we start looping over each of the individual bounding boxes, scores, and predicted labels. If the `score` is below the minimum confidence, we ignore it, thereby filtering out weak predictions.

Keep in mind that the TFOD API requires our bounding boxes to be in the range $[0, 1]$ — to draw them on our image, we first need to call them back to the range $[0, W]$ and $[0, H]$, respectively (**Lines 100-104**).

Lines 107-114 handle looping up the human readable `label` in the `categoryIdx` dictionary and then drawing the label and associated probability on the image. Finally, **Lines 117 and 118** display the output image to our screen.

To execute our `predict.py` script, open up your terminal, and execute the following command, ensuring that you supply a valid path to an input image:

```
$ python predict.py \
     --model lisa/experiments/exported_model/frozen_inference_graph.pb \
     --labels lisa/records/classes.pbtxt \
     --image path/to/input/image.png \
     --num-classes 3
```

I have executed the `predict.py` on a number of example images in the LISA testing set. A montage of these detections can be seen in Figure 16.5.

16.4 Summary

In this chapter we learned how to use the TensorFlow Object Detection API to train a Faster R-CNN + ResNet-101 architecture to detect the presence of traffic signs in images. Overall, we were able to

train a very high **98% mAP@0.5 IoU** after a total of 50,000 steps.

You'll typically want to train your networks for a bare minimum of 20,000 steps — higher accuracy can be obtained using 50,000-200,000 steps depending on the dataset.

After we trained our Faster R-CNN using the TFOD API, we then created a Python script to apply our network to our own input images. This same script can be utilized in your own projects. An example script used to apply models to input video streams/files can be found in the downloads associated with this book.

In our next chapter we'll continue the theme of deep learning for self-driving cars by training a SSD to detect the front and rear view of vehicles.

17. Single Shot Detectors (SSDs)

In our previous two chapters we discussed the Faster R-CNN framework and then trained a Faster R-CNN to detect road signs in images. While Faster R-CNNs enabled us to achieve our goal, we found two problems that needed to be addressed:

1. The framework is complex, including multiple moving parts
2. We achieved approximately $7 - 10$ FPS — reasonable for a deep learning-based object detector, but not if we want true real-time performance

To address these concerns we'll review the Single Shot Detector (SSD) framework for object detection. The SSD object detector is entirely end-to-end, contains no complex moving parts, and is capable of super real-time performance.

17.1 Understanding Single Shot Detectors (SSDs)

The SSD framework was first introduced by Liu et al. in their 2015 paper, *SSD: Single Shot MultiBox Detector* [61]. The Mulitbox component, used for the bounding box regression algorithm, comes from Szegedy (the same Christian Szegedy of Google's Inception network) et al.'s 2015 paper,*Scalable High Quality Object Detection* [62].

Combining both the Multibox regression technique along with the SSD framework, we can create an object detector that reaches comparable accuracy to Faster R-CNN and obtain up to 59+ FPS — faster FPS can be obtained using smaller, more efficient base networks as well.

In the first part of this section we'll discuss some of the motivation behind SSDs, why they were created, and the problems they address in context of object detection. From there we'll review the SSD architecture, including the concept of MultiBox priors. Finally, we'll discuss how SSDs are trained.

17.1.1 Motivation

The work of Girchick et al. in their three R-CNN publications [43, 46, 51] is truly remarkable and has enabled deep learning-based object detection to become a reality. However, there are a few problems that both researchers and practitioners found with R-CNNs.

The first is that training required multiple phases. The Region Proposal Network (RPN) needed to be trained in order to generate suggested bounding boxes before we could train the actual classifier to recognize objects in images. The problem was later mitigated by training the entire R-CNN architecture end-to-end, but prior to this discovery, it introduced a tedious pre-training process.

The second issue is that training took too long. A (Faster) R-CNN consists of multiple components, including:

1. A Region Proposal Network
2. A ROI Pooling Module
3. The final classifier

While all three fit together into a framework, they are still moving parts that slow down the entire training procedure.

The final issue, and arguably most important, is that inference time was too slow — *we could not yet obtain real-time object detection with deep learning*.

In fact, we can see how SSDs attempt to address each of these problems by inspecting the name itself. The term **Single Shot** implies that both *localization* and *detection* are performed in a single forward pass of the network during inference time — the network only has to "look" at the image once to obtain final predictions.

It's important to understand the significance of the term single shot. Unlike R-CNNs that require refetching pixels from the original image or slices from the feature map, SSDs instead continue to propagate the feature maps forward, connecting the feature maps in a novel way such that objects of various sizes and scales can be detected. As we'll see, and according to Liu et al, the fundamental improvement in speed of SSDs comes from eliminating bounding box proposals and subsampling of pixels or features [61].

The term **Multibox** refers to Szegedy et al's original Multibox algorithm used for bounding box regression [62]. This algorithm enables SSDs to localize objects of different classes, even if their bounding boxes overlap. In many object detection algorithms, overlapping objects of different classes are often suppressed into a single bounding box with the highest confidence.

Finally, the term **Detector** implies that we'll be not only localizing the (x, y)-coordinates of a set of objects in an image, but also returning their class labels as well.

17.1.2 Architecture

Just like a Faster R-CNN, an SSD starts with a **base network**. This network is typically pre-trained, normally on a large dataset such as ImageNet, enabling it to learn a rich set of discriminative features. Again, we'll use this network for transfer learning, propagating an input image to a pre-specified layer, obtaining the feature map, and then moving forward to the object detection layers.

In the work of Liu et al., VGG16 was used [17], as at the time of publication, VGG was found to provide better transfer learning accuracy/results than other popular network architectures. Today we would instead use the deeper ResNet [21, 22] or DenseNet [55] architectures to obtain higher accuracy, or we may swap in SqueezeNet [30] or MobileNet [53] for additional speed. For the sake of the explanation of the SSD framework in this section, we'll use VGG as our base network.

Figure 17.1 illustrates the SSD architecture. We utilize the VGG layers up until `conv_6` and then detach all other layers, including the fully-connected layers. A set of new `CONV` layers are then added to the architecture — these are the layers that make the SSD framework possible. As you can see from the diagram, each of these layers are *CONV* layers as well. This behavior implies that our network is fully-convolutional: we can accept an input image of arbitrary size — we are no longer restricted by the 224×224 input requirements of VGG.

We'll review what these additional auxiliary layers are doing later in this chapter, but for the

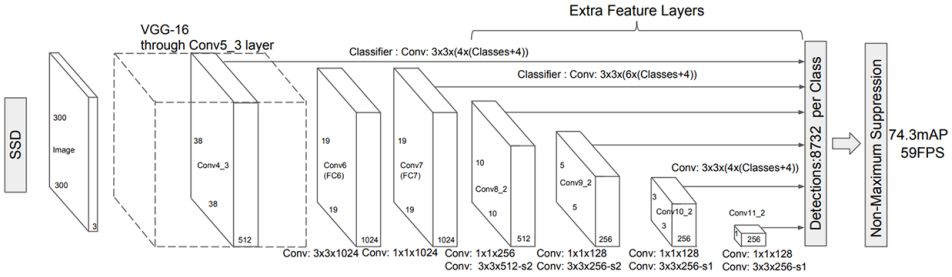


Figure 17.1: A diagram of the Single Shot Detector (SSD) architecture. The SSD starts with a base network (typically pre-trained on ImageNet). A set of new `CONV` layers are used to *replace* later `CONV` and `POOL` layers. Each `CONV` layer connects to the output `FC` layer. Combined with the (modified) Multibox algorithm, this allows SSDs to detect objects at varying scales in the image in a single forward pass. (Credit: Figure 2 of Lui et al. [61])

time being notice two important components:

1. We progressively reduce the volume size in deeper layers, as we would with a standard CNN
2. Each of the `CONV` layers connects to the final detection layer.

The fact that each feature map connects to the final detection layer is important — it allows the network to detect and localize objects in images at varying scales. Furthermore, this scale localization happens in a forward pass. No resample of feature maps is required, enabling SSDs to operate in an entire feedforward manner — this fact is what makes SSDs so fast and efficient.

17.1.3 MultiBox, Priors, and Fixed Priors

The SSD framework uses a modified version of Szegedy et al.'s MultiBox algorithm [62] for bounding box proposals. This algorithm is inspired by Szegedy's previous work on the Inception network — MultiBox uses both (1) a series of 1×1 kernels to help reduce dimensionality (in terms of volume width and height) and (2) a series of 3×3 kernels for more feature-rich learning.

The MultiBox algorithm starts with **priors**, similar to though not exactly like anchors from the Faster R-CNN framework. The priors are fixed size bounding boxes whose dimensions have been *pre-computed* based on the dimensions and locations of the ground-truth bounding boxes for each class in the dataset.

We call these a "prior" as we're relying on Bayesian statistical inference, or more specifically, *a prior probability distribution*, of where object locations will appear in an image. The priors are selected such that their Intersection over Union (IoU) is greater than 50% with ground-truth objects. It turns out that this method of computing priors is better than randomly selecting coordinates from the input image; however, the problem is that we now need to pre-train the MultiBox predictor, undermining our goal of training a complete deep learning-based object detector end-to-end.

Luckily, there is a solution: **fixed priors** — and it involves a similar anchor selection technique as Faster R-CNNs.

To visualize the concept of fixed priors, take a look at Figure 17.2. On the *left* we have our original input image with ground-truth bounding boxes. Our goal is to generate feature maps that discretizes the input image into cells (*middle* and *right*) — this process will naturally happen as the image progresses through the CNN and the output spatial dimensions of our volumes become progressively smaller and smaller.

Each cell in a feature map, similar to an anchor, has a small set of default bounding boxes (four of them) of varying aspect ratios. By discretizing the input image into different sized feature maps we are able to detect objects at different scales in the image.

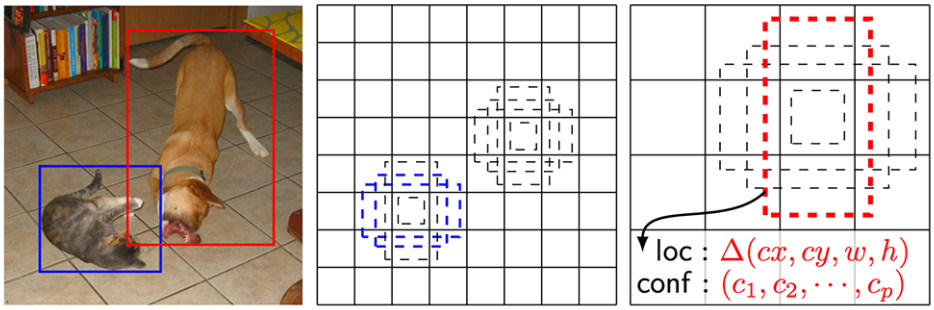


Figure 17.2: **Left:** The original input image with ground-truth bounding boxes. **Middle and Right:** Generating feature map that discretizes the input image into cells of different sizes. Each cell (similar to an anchor in Faster R-CNN terminology) has a set of default bounding boxes associated with it. The *smaller* the number of cells, the *larger* the object that can be detected. The *larger* the number of cells, the *smaller* the object can be detected. (Credit: Figure 1 of Lui et al. [61])

To understand this process further, consider the middle 8×8 feature map. Here our surrounding bounding box priors are small, capable of localizing small objects. However, in the right 4×4 feature map our bounding box priors are larger, capable of localizing larger objects.

In the example from Liu et al., the 8×8 feature map, in particular the *blue* highlighted bounding box prior, can be used to localize the cat in the image while the 4×4 feature map with the *red* highlighted bounding box prior can localize the dog — the dog is much larger than the cat and thus requires a discretized feature map with fewer cells.

This process of discretization of feature maps (whose spatial dimensions are subsequently reduced later in the CNN) combined with varying aspect ratios allows us to efficiently localize objects of varying scales, viewpoints, and ratios.

Furthermore, just like in Faster R-CNN, instead of predicting raw (x, y) coordinates, we predict the bounding box offsets (i.e., deltas) for each class label.

To make this point more clear, consider a bounding box consisting of two sets of (x, y)-coordinates with b default fixed priors per feature map cell along with c total class labels. If our feature map spatial dimensions are $f = M \times N$, then we would compute a total of $t = f \times b \times (4 + c)$ values for each feature map [61, 63].

Again, it's important to keep in mind that for each predicted bounding box we are also computing the probability of all class labels inside the region rather than keeping only the bounding box with the largest probability across all classes. Computing, and retaining, the probability for the bounding boxes in a class-wise manner enables us to detect potentially overlapping objects as well.

17.1.4 Training Methods

When training a SSD we need to consider the loss function of the MultiBox algorithm which includes two components:

1. Confidence loss
2. Location loss

Categorical cross-entropy loss is used for confidence, as it measures if we were correct in our class label prediction for the bounding box. The location loss, similar to Faster R-CNN, uses a smooth L1-loss, giving the SSD more leeway for close but not perfect localizations (i.e., the final predicted bounding box does not need to be perfect just "close enough"). Keep in mind that predicting bounding boxes that match the ground-truth is simply unrealistic.

Before training your network you'll also want to consider the number of default bounding boxes

used by the SSD — the original paper by Liu et al. recommends using four or six default bounding boxes. You may adjust these numbers as you see fit, but you'll typically want to keep them at the default recommended values. Adding more variations in scale and aspect ratio can potentially (but not always) enable you to detect more objects, but will also significantly slow down your inference time.

The same concept is true for the number of feature maps as well — you can include additional `CONV` blocks to the network, therefore increasing depth which leads to increasing the likelihood of an object being correctly detected and classified. The tradeoff here is again speed. The more `CONV` layers you add, the slower the network will run.

Liu et al. provide an extensive study of the tradeoffs between adding/removing the number of default bounding boxes and `CONV` blocks in their paper [61] — please refer to it if you are interested in more details on these tradeoffs.

The SSD framework also includes a common object detection concept of **hard-negative mining** to increase training accuracy. During the training process, cells that have a low IoU with ground-truth objects are treated as negative examples.

By definition, most objects in an image will be contained in a small fraction of the image. It's therefore possible that most of the image will be considered *negative* examples — if we used all negative examples for an input image, our proportions of positive to negative training examples would be *extremely* unbalanced.

To ensure the number of negative examples do not overwhelm the number of positive ones, Liu et al. recommends keeping a ratio of negative to positive examples around 3:1. Most SSD implementations you will use either do this sampling by default or provide it as a tunable parameter for the network.

The SGD optimizer was used in the original paper to train end-to-end, but other optimizers can be used. Typically, you will see see RMSprop or Adam, especially when fine-tuning an existing object detection model.

During prediction time, non-maxima suppression is used class-wise, yielding the final predictions from the network. Once trained, Liu et al. demonstrated SSDs obtained comparable accuracy to their Faster R-CNN counterparts on a number of object detection datasets, including PASCAL VOC [58], COCO [59], and ILSVRC [9].

However, the major contribution here is that SSDs can run at approximately 22 FPS on 512×512 input images and 59 FPS on 300×300 images. Even higher FPS can be obtained by swapping out the VGG base network for a more computationally efficient MobileNet (although accuracy may decrease slightly).

17.2 Summary

In this chapter we reviewed the fundamentals of the Single Shot Detector (SSD) framework. Unlike Faster R-CNNs which contain multiple moving parts and components, SSDs are unified, encapsulated in a single end-to-end network, making SSDs easier to train and capable of real-time object detection while still obtaining comparable accuracy.

The primary criticism of SSDs is that they tend to not work well for small objects, mainly because small objects may not appear on all feature maps — the more an object appears on a feature map, the more likely that the MultiBox algorithm can detect it.

A common hack to address the problem is to increase the size of the input image, but that (1) reduces the speed at which SSD can run and (2) does not completely alleviate the problem of detecting small objects. If you are trying to detect objects that are small relative to the size of the input image you should consider using Faster R-CNN instead. For further information on SSDs, refer to Liu et al.'s publication [61] along with Eddie Forson's excellent introductory article [63].

Continuing our theme of deep learning for self-driving cars, in our next chapter we'll learn how to train an SSD from scratch to recognize the front and rear view of vehicles using Google's Object Detection API.

18. Training a SSD From Scratch

In our last chapter we discussed the inner workings of the Single Shot Detector (SSD) from Liu et al. Now that we have an understanding of the SSD framework we can use the TensorFlow Object Detection API (TFOD API) to train a SSD on dataset, similar to Chapter 16 where we trained a Faster R-CNN using the TFOD API.

We'll be continuing the theme of deep learning for self-driving cars by training a SSD to recognize both the *front* and *rear* views of vehicles using a vehicle dataset curated and labeled by Davis King of dlib [40]. Not only will this process give us increased exposure to working with the TFOD API and toolkit, but it will also give us additional insight into the limitations of the TFOD API and how certain types of datasets can be more challenging to train than others due to hard-negative mining.

18.1 The Vehicle Dataset

The dataset we are using for this example comes from Davis King's dlib library [40] and was hand annotated by King for usage in a demonstration of his max-margin object detection algorithm [64, 65].

Each image in this dataset was captured from a camera mounted to a car's dashboard. For each image, all visible front and rear views of vehicles are labeled as such. An example of such vehicle annotations in the dataset can be found in Figure 18.1.

The images were labeled using the `imglab` tool included with dlib [66]. One of the benefits of using `imglab` for annotation is that we can mark potentially "confusing" areas of an image as "ignore". Provided you are using the dlib library to train your own HOG + Linear SVM detector or CNN MMOD detector (using C++), dlib will *exclude* the areas marked as ignored from training.

Examples of areas that can and should be marked as ignored in context of vehicle detection include:

1. Areas where a large number of cars are in a compact area where it's not clear where one car begins and the other ends
2. Vehicles at a distance that are too small to be 100% visually recognizable, but the CNN can still "see" them and learn patterns from them

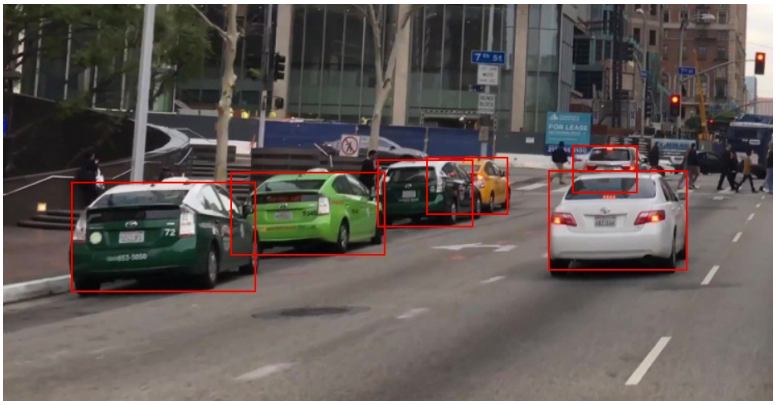


Figure 18.1: An example of rear views of vehicles and their associated bounding boxes in the dlib Vehicle dataset [67].

Unfortunately, the TFOD API has no concept of "ignored" areas of images which can hurt our object detection performance. The goal of using dlib's vehicle dataset is twofold:

1. To train a high quality SSD object detector to localize front and rear views of vehicles in images
2. To demonstrate how a combination of (1) SSD's hard-negative mining with (2) the lack of "ignored" areas in the TFOD API can confuse our object detector

By the end of this chapter you'll have a strong understanding of both goals, enabling you to make intelligent decisions regarding which object detection framework to use when training your own detectors on custom datasets.

18.2 Training Your SSD

This section will demonstrate how to train a SSD using the TFOD API, similar to how we trained a Faster R-CNN using the TFOD API in Chapter 16. Since we have already reviewed the TFOD API in detail, we'll be spending less time discussing the API and associated tools here. For a more detailed review of the TFOD API and toolkit, please refer to Chapter 16.

18.2.1 Directory Structure and Configuration

Similar to Chapter 16 where we trained a Faster R-CNN, we'll have a near identical directory structure for this project:

```
| --- ssds_and_rcnn
|     |--- build_vehicle_records.py
|     |--- config
|     |     |--- __init__.py
|     |     |--- dlib_front_rear_config.py
|     |--- dlib_front_and_rear_vehicles_v1/
|     |     |--- image_metadata_stylesheet.xsl
|     |     |--- input_videos
...
|     |     |--- testing.xml
|     |     |--- training.xml
|     |     |--- youtube_frames
|     |--- predict.py
|     |--- predict_video.py
```

We'll create a module named `config` where we'll store all necessary Python-based configurations inside `dlib_front_rear_config.py`.

The `build_vehicle_records.py` will be used to build the vehicles dataset in TensorFlow record format. As we'll see, `build_vehicle_records.py` is very similar to `build_lisa_records.py`, only with a handful of modifications.

Both `predict.py` and `predict_video.py` are identical to Chapter 16. The `setup.sh` script will be used to configure our `PYTHONPATH` to access the TFOD API imports and libraries, again, identical to Chapter 16.

The `dlib_front_and_rear_vehicles_v1` directory contain the vehicles dataset. You can download the dataset using the following link (http://pyimg.co/9gq7u) and then clicking the `dlib_front_and_rear_vehicles_v1.tar` file.

Alternatively, you can use `wget` and `tar` to download and unarchive the files:

```
$ wget http://dlib.net/files/data/dlib_front_and_rear_vehicles_v1.tar
$ tar -xvf dlib_front_and_rear_vehicles_v1.tar
```

Inside `dlib_front_and_rear_vehicles_v1` you'll find your image files, along with two XML files, `training.xml` and `testing.xml`:

```
$ cd dlib_front_and_rear_vehicles_v1
$ ls *.xml
testing.xml  training.xml
```

These files contain our bounding box annotations + class labels for both the training and testing set, respectively. You'll want to take the time now to create your `records` and `data` directories as we did in Chapter 16:

```
$ mkdir records experiments
$ mkdir experiments/training experiments/evaluation experiments/exported_model
```

Creating these directories ensures that our project structure matches Chapter 16 on Faster R-CNN, enabling us to reuse the majority of our code and example commands.

Go ahead and open up `dlib_front_rear_config.py` and we'll review the contents:

```
1  # import the necessary packages
2  import os
3
4  # initialize the base path for the front/rear vehicle dataset
5  BASE_PATH = "dlib_front_and_rear_vehicles_v1"
6
7  # build the path to the input training and testing XML files
8  TRAIN_XML = os.path.sep.join([BASE_PATH, "training.xml"])
9  TEST_XML = os.path.sep.join([BASE_PATH, "testing.xml"])
```

Line 5 defines the `BASE_PATH` to the vehicles dataset on our system. We then define `TRAIN_XML` and `TEST_XML`, the paths to the training and testing files provided with the vehicles dataset, respectively.

If you were to open one of these files you'll notice that it's an XML file. Each `image` element in the file includes the `file` path along with a number of bounding `box` objects. We'll learn how to parse this XML file in Section 18.2.2.

From there, we define the path to our record files:

```
# build the path to the output training and testing record files,
# along with the class labels file
TRAIN_RECORD = os.path.sep.join([BASE_PATH,
     "records/training.record"])
TEST_RECORD = os.path.sep.join([BASE_PATH,
     "records/testing.record"])
CLASSES_FILE = os.path.sep.join([BASE_PATH,
     "records/classes.pbtxt"])
```

Along with the class labels dictionary:

```
# initialize the class labels dictionary
CLASSES = {"rear": 1, "front": 2}
```

Again, keeping in mind that the label ID `0` is reserved for the *background* class (hence why we start counting from `1` rather than `0`).

18.2.2 Building the Vehicle Dataset

The vast majority of the `build_vehicle_records.py` file is based on `build_lisa_records.py` from Chapter 16 — the primary modification is parsing the vehicle dataset XML file rather than the CSV file provided with the LISA Traffic Sign dataset.

There are a number of libraries used to parse XML files, but my favorite one is BeautifulSoup [68]. If you do not have BeautifulSoup installed on your system, you can install it via `pip`:

```
$ pip install beautifulsoup4
```

If you are using Python virtual environments make sure you use the `workon` command to access your respective Python virtual environment before installing BeautifulSoup. I will assume that you understand the extreme basics of XML files, including tags and attributes — advanced knowledge of XML files is certainly not required.

If you have never used XML files before, I suggest you read the following tutorial to help you get up to speed before continuing: http://pyimg.co/5wzvd

Let's go ahead and get started — open up `build_vehicle_records.py` and insert the following code:

```
# import the necessary packages
from config import dlib_front_rear_config as config
from pyimagesearch.utils.tfannotation import TFAnnotation
from bs4 import BeautifulSoup
from PIL import Image
import tensorflow as tf
import os
```

Line 2 imports our configuration file so we can access it throughout the Python script. Our `TFAnnotation` class (**Line 3**) will enable us to efficiently build TensorFlow data points that will be written to our output record file. We'll also import `BeautifulSoup` so we can parse our image paths, bounding boxes, and class labels from our XML files.

Our next code block handles writing the `CLASSES` to file in the JSON/YAML-like format that TensorFlow expects (which is again identical to Chapter 16):

```
9  def main(_):
10     # open the classes output file
11     f = open(config.CLASSES_FILE, "w")
12
13     # loop over the classes
14     for (k, v) in config.CLASSES.items():
15         # construct the class information and write to file
16         item = ("item {\n"
17                 "\tid: " + str(v) + "\n"
18                 "\tname: '" + k + "'\n"
19                 "}\n")
20         f.write(item)
21
22     # close the output classes file
23     f.close()
24
25     # initialize the data split files
26     datasets = [
27         ("train", config.TRAIN_XML, config.TRAIN_RECORD),
28         ("test", config.TEST_XML, config.TEST_RECORD)
29     ]
```

Lines 26-29 defines a `datasets` tuple consisting of the *input* training/testing XML file and the *output* training/testing record file. In Chapter 16 we had to manually build our training and testing split, but for the vehicles dataset our data is pre-split — no additional work is required from us on this front.

Next, let's start looping over each of the training and testing splits, respectively:

```
31     # loop over the datasets
32     for (dType, inputPath, outputPath) in datasets:
33         # build the soup
34         print("[INFO] processing '{}'...".format(dType))
35         contents = open(inputPath).read()
36         soup = BeautifulSoup(contents, "html.parser")
37
38         # initialize the TensorFlow writer and initialize the total
39         # number of examples written to file
40         writer = tf.python_io.TFRecordWriter(outputPath)
41         total = 0
```

Line 35 loads the contents of the current `inputPath` while **Line 36** builds the `soup` by applying BeautifulSoup to parse the contents and build an XML tree that we can easily traverse in memory.

Line 40 creates a `writer`, an instantiation of `TFRecordWriter`, that we can use to write images and bounding boxes to the output TensorFlow record file.

Let's traverse the XML document by looping over all `image` elements:

```
43          # loop over all image elements
44          for image in soup.find_all("image"):
45               # load the input image from disk as a TensorFlow object
46               p = os.path.sep.join([config.BASE_PATH, image["file"]])
47               encoded = tf.gfile.GFile(p, "rb").read()
48               encoded = bytes(encoded)
49
50               # load the image from disk again, this time as a PIL
51               # object
52               pilImage = Image.open(p)
53               (w, h) = pilImage.size[:2]
54
55               # parse the filename and encoding from the input path
56               filename = image["file"].split(os.path.sep)[-1]
57               encoding = filename[filename.rfind(".") + 1:]
58
59               # initialize the annotation object used to store
60               # information regarding the bounding box + labels
61               tfAnnot = TFAnnotation()
62               tfAnnot.image = encoded
63               tfAnnot.encoding = encoding
64               tfAnnot.filename = filename
65               tfAnnot.width = w
66               tfAnnot.height = h
```

For each of the `image` elements, we exact the file path attribute (**Line 46**). **Lines 47 and 48** load the image from disk in the `encoded` TensorFlow format. **Lines 52 and 53** load the image from disk again, this time in PIL/Pillow format, enabling us to extract the image dimensions.

Lines 56 and 57 extract the `filename` from the image path and then use the `filename` to derive the image `encoding` (i.e., JPG, PNG, etc.). From there we can initialize our `tfAnnot` object (**Lines 61-66**). For a more detailed review of this code, please see Section 16.3.3 in the Faster R-CNN chapter.

Each `image` element in the XML tree also has a series of children bounding `box` elements — let's loop over all `box` elements for a particular `image` now:

```
68               # loop over all bounding boxes associated with the image
69               for box in image.find_all("box"):
70                    # check to see if the bounding box should be ignored
71                    if box.has_attr("ignore"):
72                         continue
73
74                    # extract the bounding box information + label,
75                    # ensuring that all bounding box dimensions fit
76                    # inside the image
77                    startX = max(0, float(box["left"]))
78                    startY = max(0, float(box["top"]))
79                    endX = min(w, float(box["width"]) + startX)
80                    endY = min(h, float(box["height"]) + startY)
81                    label = box.find("label").text
```

Line 71 makes a check to see if the `ignore` attribute of the `box` is set, and if so, we do not further process the bounding box. In an ideal world, the TFOD API would allow us to mark specific

ROIs as "ignore", like in dlib, thereby instructing our network not to be trained on certain areas of the image. Unfortunately, this process of "ignoring" certain regions of the image is not possible with the TFOD API.

Lines 77-81 extract the bounding box and class label information from the `box` element. The bounding box coordinates in our respective XML files can be negative or larger than the actual width and height of the image — we take special precautions to ensure we clip these values using the `max` and `min` functions, respectively.

So, why exactly does our dataset have bounding box coordinates that are outside the dimensions of the image? The reason lies in how the dlib library and `imglab` tool work.

The dlib library requires that objects have similar bounding box aspect ratios during training — if bounding boxes *do not* have similar aspect ratios, the algorithm will error out. Since vehicles can appear at the borders of an image, in order to maintain similar aspect ratios, the bounding boxes can actually extend *outside* the boundaries of the image. The aspect ratio issue isn't a problem with the TFOD API so we simply clip those values.

Next, we can scale the bounding box coordinates to the range $[0, 1]$, which is what TensorFlow expects:

```
83                  # TensorFlow assumes all bounding boxes are in the
84                  # range [0, 1] so we need to scale them
85                  xMin = startX / w
86                  xMax = endX / w
87                  yMin = startY / h
88                  yMax = endY / h
```

Our next code block makes a check to ensure that our maximum x and y values are always larger than their corresponding minimums (and vice versa):

```
90                  # due to errors in annotation, it may be possible
91                  # that the minimum values are larger than the maximum
92                  # values -- in this case, treat it as an error during
93                  # annotation and ignore the bounding box
94                  if xMin > xMax or yMin > yMax:
95                      continue
96
97                  # similarly, we could run into the opposite case
98                  # where the max values are smaller than the minimum
99                  # values
100                 elif xMax < xMin or yMax < yMin:
101                     continue
```

If either of these checks fail, we ignore the bounding box and presume it to be an error during the annotation process.

When I first trained a network on the vehicles dataset using the TFOD API, I spent over a day trying to diagnose why my network errored out during the training process. It turns out the problem was related to bounding boxes having invalid coordinates where the maximum x or y value was actually *smaller* than the corresponding minimum (and vice versa). Applying these checks on **Lines 90-101** discarded the invalid bounding boxes (two of them) and allowed the network to be trained.

From there we can continue building our `tfAnnot` object and add it to our output record file:

```
                    # update the bounding boxes + labels lists
                    tfAnnot.xMins.append(xMin)
                    tfAnnot.xMaxs.append(xMax)
                    tfAnnot.yMins.append(yMin)
                    tfAnnot.yMaxs.append(yMax)
                    tfAnnot.textLabels.append(label.encode("utf8"))
                    tfAnnot.classes.append(config.CLASSES[label])
                    tfAnnot.difficult.append(0)

                    # increment the total number of examples
                    total += 1

                # encode the data point attributes using the TensorFlow
                # helper functions
                features = tf.train.Features(feature=tfAnnot.build())
                example = tf.train.Example(features=features)

                # add the example to the writer
                writer.write(example.SerializeToString())
```

Our final code block closes the `writer` object and starts the main thread of execution:

```
        # close the writer and print diagnostic information to the
        # user
        writer.close()
        print("[INFO] {} examples saved for '{}'".format(total,
            dType))

# check to see if the main thread should be started
if __name__ == "__main__":
    tf.app.run()
```

To build the vehicle dataset, open up a terminal, ensure you have executed `setup.sh` to configure your `PYTHONPATH`, and execute the following command:

```
$ time python build_vehicle_records.py
[INFO] processing 'train'...
[INFO] 6133 examples saved for 'train'
[INFO] processing 'test'...
[INFO] 382 examples saved for 'test'

real    0m2.749s
user    0m2.411s
sys     0m1.163s
```

Listing the contents of my `records` directory we can see that our training and testing record files have been successfully created:

```
$ ls dlib_front_and_rear_vehicles_v1/records/
classes.pbtxt  testing.record  training.record
```

18.2.3 Training the SSD

In order to train our SSD on the vehicles dataset, we first need to head back to TensorFlow Object Detection Model Zoo (http://pyimg.co/1z34r) and download the SSD + Inception v2 trained on COCO. Figure 18.2 shows a screenshot of the two SSDs available to us:

- SSD + MobileNet
- SSD + Inception

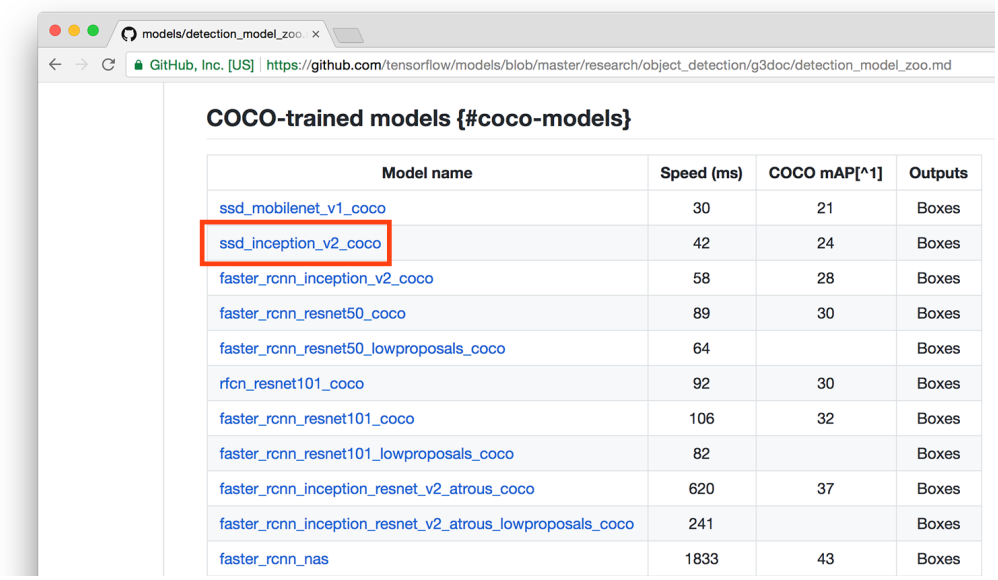


Figure 18.2: We will be training a SSD with Inception base network for vehicle detection. Make sure you download the SSD + Inception model weights pre-trained on COCO from the TensorFlow Model Zoo.

We'll be using the SSD + Inception architecture in this chapter as it yields higher detection accuracy, but you can use SSD + MobileNet if you so desire. Go ahead and download the SSD + Inception network using the link above or use `wget` and `tar` to download and unarchive the archive, which at the time of this writing is the November 17 2017, version (`ssd_inception_v2_coco_2017_11_17.tar.gz`):

```
$ cd dlib_front_and_rear_vehicles_v1/experiments/training
$ mv ~/Downloads/ssd_inception_v2_coco_2017_11_17.tar.gz ./
$ tar -zxvf ssd_inception_v2_coco_2017_11_17.tar.gz
$ ls -l ssd_inception_v2_coco_2017_11_17
checkpoint
frozen_inference_graph.pb
model.ckpt.data-00000-of-00001
model.ckpt.index
model.ckpt.meta
saved_model
```

Next, we need our base configuration file. Head back to the sample configuration pages (http://pyimg.co/r2xql) and download the `ssd_inception_v2_pets.config` file.

Once you have downloaded the configuration file, move it to our dlib Vehicles dataset directory and rename it:

```
$ cd dlib_front_and_rear_vehicles_v1/experiments/training
$ mv ~/Downloads/ssd_inception_v2_pets.config ssd_vehicles.config
```

As you can see from the commands above, I have renamed the file `ssd_vehicles.config`.

Just as we updated the configuration in Chapter 16 on Faster R-CNN, we need to do the same for our SSD. The first change is to set `num_classes` to 2, as we have two class labels, the "front" and "rear" views of the vehicles, respectively:

```
7  model {
8    ssd {
9      num_classes: 2
10     box_coder {
11       faster_rcnn_box_coder {
12         y_scale: 10.0
13         x_scale: 10.0
14         height_scale: 5.0
15         width_scale: 5.0
16       }
17     }
```

You'll then want to set the `fine_tune_checkpoint` in `train_config` to point to your downloaded SSD + Inception model checkpoint:

```
150    fine_tune_checkpoint: "/home/adrian/pyimagesearch/dlbook/ssds_and_rcnn/
151       dlib_front_and_rear_vehicles_v1/experiments/training/
152       ssd_inception_v2_coco_2017_11_17/model.ckpt"
153    from_detection_checkpoint: true
```

Keep in mind that your path will be different than mine so take a few minutes to validate that your path is correct — this step will save you some headaches down the line.

I'm going to allow my model to train for a maximum of 200,000 steps; however, you could obtain approximately the same accuracy at 75,000-100,000 steps:

```
134  train_config: {
135    batch_size: 24
136    ...
137    num_steps: 200000
138  ...
139  }
```

Next we'll update our `train_input_reader` to point to our `training.record` and `classes.pbtxt` file:

```
167  train_input_reader: {
168    tf_record_input_reader {
169      input_path: "/home/adrian/pyimagesearch/dlbook/ssds_and_rcnn/
170            dlib_front_and_rear_vehicles_v1/records/training.record"
171    }
172    label_map_path: "/home/adrian/pyimagesearch/dlbook/ssds_and_rcnn/
173            dlib_front_and_rear_vehicles_v1/records/classes.pbtxt"
174  }
```

As well as our `eval_input_reader` to point to the `testing.record` and `classes.pbtxt` file:

```
eval_input_reader: {
  tf_record_input_reader {
    input_path: "/home/adrian/pyimagesearch/dlbook/ssds_and_rcnn/
          dlib_front_and_rear_vehicles_v1/records/training.record"
  }
  label_map_path: "/home/adrian/pyimagesearch/dlbook/ssds_and_rcnn/
          dlib_front_and_rear_vehicles_v1/records/classes.pbtxt"
  shuffle: false
  num_readers: 1
}
```

Again, I would recommend updating your `eval_config` to either use the full `num_examples` (if using a GPU) or `5-10` if you are using a CPU:

```
eval_config: {
  num_examples: 382
  # Note: The below line limits the evaluation process to 10 evaluations.
  # Remove the below line to evaluate indefinitely.
  #max_evals: 10
}
```

Provided you have properly set your file paths, you should be all set! Open up a new terminal (ensuring you execute `setup.sh` from the `ssds_and_rcnn` directory to set your `PYTHONPATH`) and change directory to `models/research`, and execute the following command:

```
$ source setup.sh
$ cd ~/models/research
$ python object_detection/train.py --logtostderr \
    --pipeline ~/ssds_and_rcnn/dlib_front_and_rear_vehicles_v1/
        experiments/training/ssd_vehicles.config \
    --train_dir ~/ssds_and_rcnn/dlib_front_and_rear_vehicles_v1/
        experiments/training
INFO:tensorflow:Starting Queues.
INFO:tensorflow:global_step/sec: 0
INFO:tensorflow:Recording summary at step 0.
INFO:tensorflow:global step 1: loss = 18.8345 (14.528 sec/step)
INFO:tensorflow:global step 2: loss = 17.7939 (0.966 sec/step)
INFO:tensorflow:global step 3: loss = 17.3796 (0.885 sec/step)
...
```

The training process has now started, but we also need our evaluation script

```
$ python object_detection/eval.py --logtostderr \
    --pipeline_config_path ~/ssds_and_rcnn/dlib_front_and_rear_vehicles_v1/
        experiments/training/ssd_vehicles.config \
    --checkpoint_dir ~/ssds_and_rcnn/dlib_front_and_rear_vehicles_v1/
        experiments/training \
    --eval_dir ~/ssds_and_rcnn/dlib_front_and_rear_vehicles_v1/experiments
```

As well as our TensorBoard dashboard:

```
$ cd ~/pyimagesearch/dlbook/ssds_and_rcnn/
$ tensorboard --logdir dlib_front_and_rear_vehicles_v1/experiments
TensorBoard 0.4.0rc3 at http://annalee:6006 (Press CTRL+C to quit)
```

The paths to the input files and directories can be quite long, so for convenience, I have created a sym-link from the `ssds_and_rcnn` directory to the `models/research` directory, enabling me to spend less time typing and debugging long file paths. You may wish to apply this tactic as well.

On my machine, I can visualize the training process by opening up a web browser and pointing it to `http://annalee:6006`. On your machine the port should still be `6006` but the hostname (i.e., `annalee`) will be either the IP address of your machine or the name of the machine itself.

Again, make sure you have properly `source`'d your `setup.sh` file to set your `PYTHONPATH` before executing any of the above commands.

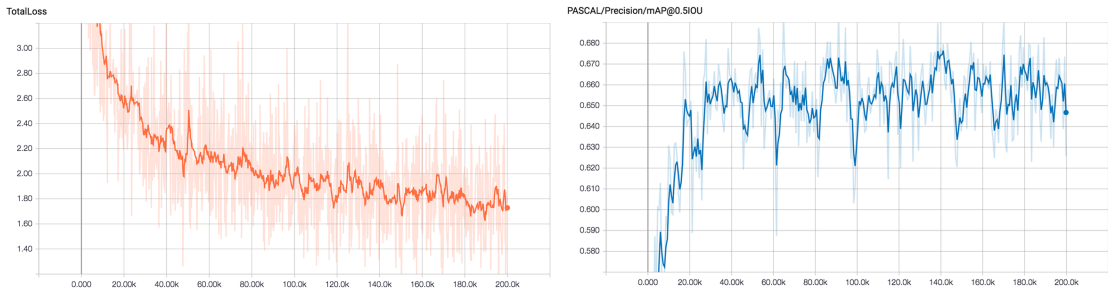


Figure 18.3: **Left:** Total loss when training SSD + Inception on dlib Vehicles dataset. Loss starts off high, then gradually decreases over the course of 200,000 steps. Further training yields marginal gains in accuracy and loss. **Right:** Total mAP@0.5 for both the *front* and *rear* labels. We reach $\approx 65.5\%$ accuracy, a respectable accuracy for object detection.

Figure 18.3 shows my training process over 200,000 steps, which took approximately 50 hours on a single GPU. The loss ended at approximately 1.7 with the mAP@0.5 around 65.5%. Further training would only marginally improve the results, if it all. Ideally, I would like to see the loss below 1 and *certainly* below 1.5 — we'll save a discussion regarding why we can't push loss that low for later in Section 18.2.5.

18.2.4 SSD Results

Now that our SSD has been trained, let's export it:

```
$ cd ~/models/research
$ python object_detection/export_inference_graph.py --input_type image_tensor \
    --pipeline_config_path ~/ssds_and_rcnn/dlib_front_and_rear_vehicles_v1/experiments/
        training/ssd_vehicles.config \
    --trained_checkpoint_prefix ~/ssds_and_rcnn/dlib_front_and_rear_vehicles_v1/
        experiments/training/model.ckpt-200000 \
    --output ~/ssds_and_rcnn/dlib_front_and_rear_vehicles_v1/experiments/exported_model
```

Here you can see that I have exported the model checkpoint at step 200,000 to the `exported_model` directory; however, you could obtain approximately the same accuracy at 75,000-100,000 steps:

Now that our model is exported, we can apply it to an example image, ensuring that you supply a valid path to an input image:

```
$ python predict.py \
    --model dlib_front_and_rear_vehicles_v1/experiments/
        exported_model/frozen_inference_graph.pb \
    --labels dlib_front_and_rear_vehicles_v1/records/classes.pbtxt \
    --image path/to/input/image.png \
    --num-classes 2
```

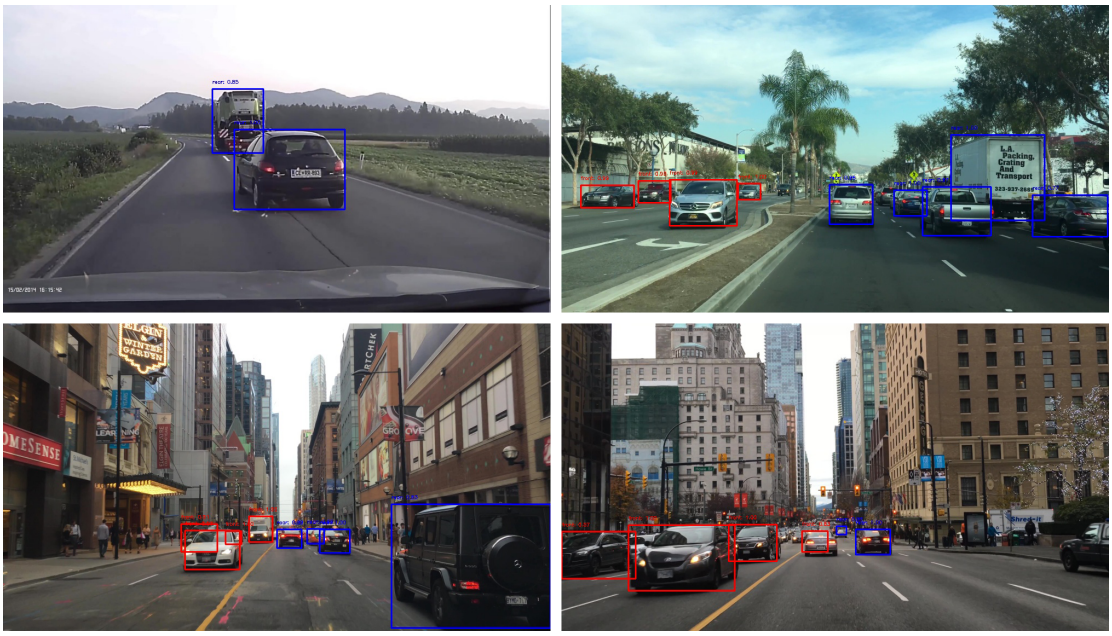


Figure 18.4: Examples of our SSD + Inception network classifying input testing frames from the dlib vehicle dataset. Our architecture does an extremely good job of detection + labeling, except for the *bottom-right* where we misclassify a rear view of a car as a frontal view. In some cases, no matter how much you tune your hyperparameters, you cannot overcome limitations of your dataset (Section 18.2.5).

I have executed the `predict.py` on a number of example images in the dlib Vehicles testing set. Figure 18.4 displays a montage of these example detections. Here we see that our SSD is typically able to correctly detect and label the "front" versus "rear" view of a vehicle; however, in some cases the detection is correct, but the label is off.

Is there a problem with our detector? Should we have used a different optimizer? Do we need to tune our hyperparameters?

Sometimes, no matter how you tune your hyperparameters, you cannot overcome the limitations of your dataset and/or the framework you are using to train your object detector, which is exactly the topic of our next section.

18.2.5 Potential Problems and Limitations

There are two limitations and drawbacks to our results. The first one can be discerned from inspecting the results and reading Chapter 17, where we discussed the fundamentals of Single Shot Detectors — SSDs do not perform well on small objects.

Looking at the output of our results, you'll see that small vehicles are missed by the detector. We could help reduce this problem by increasing the resolution of our input images, but if we *really* want to detect small vehicles at a distance, we might want to consider using the Faster R-CNN framework. Of course, there is a tradeoff — if we use a Faster R-CNN we'll sacrifice the speed of our object detection pipeline.

In the context of vehicle detection and this specific project, recognizing small vehicles at a distance is unlikely to be a problem. Vehicles that are far away are not an immediate concern to us. As they approach, our SSD will be able to recognize their presence, and instruct the self-driving vehicle to take corrective action. Given that our SSD can run at over 20 FPS, our concerns of not recognizing small objects can be mitigated, but we still need to be concerned with vehicles far away operating at very high speeds that can quickly close the gap.

The second problem we can encounter is our SSD confusing "front" and "rear" views" of vehicles. These misclassifications are partially due to the fact that the front and rears of vehicles, while semantically different, still share many visually similar characteristics, but the larger problem is being unable to mark images as "ignore" with the TFOD API.

Vehicles at a distance still contain vehicle-like characteristics which can confuse the TFOD API during training. This problem is further compounded by the hard-negative mining algorithm of SSD which tries to learn negative patterns from cells that likely contain vehicles.

That said, this issue is not specific to SSDs. I encourage you to go back and train a Faster R-CNN on the vehicles dataset — you'll notice the same behavior. If you find that you need to mark regions of images as "ignore", and areas need to be ignored in the majority of images in your dataset (such is the case with our vehicles dataset), then you may want to consider using a different object detection library than the TFOD toolkit.

18.3 Summary

In this chapter we learned how to train a Single Shot Detector to recognize the front and rear views of vehicles using Davis King's [65] dataset. Overall, we were able to obtain $\approx 65.5\%$ mAP on the dataset — visually inspecting the results confirmed that predictions were good and even capable of running in real-time.

Real-time prediction is one of the major benefits in the SSD framework. The TensorFlow Object Detection Model Zoo reports that our SSD can run at approximately 24 FPS while our Faster R-CNN can only reach 9 FPS.

However, training our SSD on the vehicle dataset highlighted some problems we can encounter when training our own custom object detectors. Since the TFOD API does not have a concept of "ignore this region of the image", we cannot exclude potentially "confusing" parts of the image from the training process. Examples of such a "confusing" area can include:

1. A large number of cars in a compact area where it's not clear where one car begins and the other ends
2. Vehicles at a distance that are too small to be 100% visually recognizable, but the CNN can still "see" them and learn patterns from them

The hard-negative mining process of an SSD is especially sensitive in these situations and can actually hurt performance. In the future I hope that the TFOD API includes a method to mark bounding boxes as "ignored" as the dlib `imglab` tool does [66], but until that point, it's important to be cognizant of these limitations.

19. Conclusions

Congratulations on finishing the *ImageNet Bundle* of *Deep Learning for Computer Vision with Python!* It's been quite the journey, and I feel quite privileged and honored to have taken this journey with you. You've learned a lot over the past 15 chapters, so let's take a second and recap your newfound knowledge. Inside this bundle you've learned how to:

- Train networks in multi-GPU environments.
- Obtain and download the ImageNet dataset (along with the associated licensing assumptions that accompany the dataset and resulting trained models).
- Prepare the ImageNet dataset by compressing the images and class labels into efficiently packed record files.
- Train state-of-the-art network architectures on ImageNet, including AlexNet, VGGNet, GoogLeNet, ResNet, and SqueezeNet – and in each case, replicating the results of the original respective authors.
- Implement a "Deep Learning as a Service" system that you can use to efficiently create deep learning API endpoints (and potentially use to start your own company).
- Train a model that can recognize facial expressions and emotions in a *real-time* video stream.
- Us transfer learning and feature extraction to automatically predict and correct the orientation of an image.
- Apply fine-tuning to recognize over 164 makes and models of vehicles with over 96.54% accuracy.
- Train a deep learning model capable of predicting the age and gender of a person in a photograph.
- Train a Faster R-CNN and Single Shot Detector (SSD) for self-driving cars, including traffic road sign detection and front/rear view vehicle detection.

Each of these projects were challenging, and representative of the types of techniques you'll need to apply when performing deep learning in the real world. Furthermore, these exact same techniques, best practices, and rules of thumb are the *exact ones* used by deep learning researchers performing state-of-the-art work. Provided you've worked through all the examples in this book, I am confident that you can confer upon yourself the title *deep learning practitioner* if not *deep*

learning expert. However, to maintain this status, you'll want to continue your studies...

19.1 Where to Now?

Deep learning is a quickly evolving field with new techniques being published every day. It may feel like a daunting task to keep up with all the new research trends, so to help you out, I've created a set of resources that I *personally use* to keep myself updated. These resources can also be used to help you study deep learning further. To start, I would *highly recommend* Stanford University's excellent cs231n class, Convolutional Neural Networks for Visual Recognition:

http://cs231n.stanford.edu/

Every spring semester a new session of the class is taught with slides and notes posted on the syllabus page:

http://cs231n.stanford.edu/syllabus.html

Links to previous semester notes can also be found there as well.

For a more theoretical treatment of deep learning, *Deep Learning* by Goodfellow et al. is a must read:

http://www.deeplearningbook.org/

The book is free to read online and can be bought on Amazon here:

http://pyimg.co/v6b1i

While not *specific* to computer vision, *Deep Learning* provides more theoretical information than covered in this hands-on book. Furthermore, if you intend on applying deep learning to domains *outside* computer vision, then you'll definitely want to read through the work of Goodfellow et al.

When it comes to staying on top of recent deep learning news, I highly recommend following the /r/machinelearning and /r/deeplearning subreddits:

- https://www.reddit.com/r/MachineLearning/
- https://www.reddit.com/r/deeplearning/

I frequent these subreddits and tend to read through most posts every 24-48 hours, bookmarking the most interesting threads for me to read later. The benefit of following these subreddits is that you'll be part of an actual *community* with whom you can discuss various techniques, algorithms, and recent publications.

Two other communities I recommend are LinkedIn Groups, *Computer Vision Online* and *Computer Vision and Pattern Recognition*, respectively:

- http://pyimg.co/s1tc3
- http://pyimg.co/6ep60

While not specific to deep learning, these groups consistently contains interesting discussions and serves as a hub for anyone interested in studying computer vision at the intersection of deep learning.

When it comes to keeping up with deep learning publications, look no further than Andrej Karpathy's *Arxiv Sanity Preserver*:

http://www.arxiv-sanity.com/

This website enables you to filter pre-print deep learning publications submitted to arXiv based on a number of terms, including:

- Most recent submission
- Top most recent submissions
- Recommend papers
- Recently discussed papers

This site is an *invaluable tool* for anyone who is intending on performing research in the deep learning field.

When it comes to software, I would suggest following the GitHub repositories for both Keras and mxnet:

- https://github.com/fchollet/keras
- https://github.com/dmlc/mxnet

By following the above repositories, you to receive email notifications as the libraries (and associated functionality) is updated. Staying on top of these packages will enable you to quickly iterate your own projects based on new features coming down the pipeline. While it's always interesting to discuss the theory behind deep learning, it's no secret that I'm a proponent of actually *implementing* what you learn – following both Keras and mxnet will enable you to improve your implementation skills.

For more general purpose deep learning news delivered to your inbox on a weekly basis, I recommend *This Wild Week in AI*, curated by Denny Britz:

http://www.wildml.com/newsletter/

I would also recommend *Deep Learning Weekly*, put together by Jan Buss and Matle Baumann:

http://www.deeplearningweekly.com/

Both of these newsletters serve up content that is relevant to the larger deep learning community, but given the traction and interest of deep learning applied to computer vision, you'll see content related to computer vision in nearly every issue.

Finally, make sure you stay up to date on the PyImageSearch.com blog and follow along with new posts. While not *every* post may be related to deep learning, you'll be able to use these tutorials to better understand computer vision and apply these techniques to actual *practical, real-world* projects.

A little bit of computer vision knowledge can go a long way when studying deep learning, so if you haven't already, be sure to take a look at both *Practical Python and OpenCV* [34]:

http://pyimg.co/ppao

And the more intensive PyImageSearch Gurus course [35]:

http://pyimg.co/gurus

Both of these resources will enable you to level-up your general computer vision skills, and in turn, help you on your deep learning career.

Thank you again for allowing me to accompany you on your journey to becoming deep learning expert. If you have any questions, please email me at adrian@pyimagesearch.com. And no matter what, keep practicing and keep implementing – deep learning is part science, part art. The more you practice, the better you'll become.

Cheers,

–Adrian Rosebrock

Bibliography

[1] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). URL: http://arxiv.org/abs/1412.6980 (cited on page 13).

[2] Geoffrey Hinton. *Neural Networks for Machine Learning*. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf (cited on page 13).

[3] Kaggle Team. *Kaggle: Dogs vs. Cats*. https://www.kaggle.com/c/dogs-vs-cats (cited on page 14).

[4] Andrej Karpathy. *Tiny ImageNet Challenge*. http://cs231n.stanford.edu/project.html (cited on page 14).

[5] Tianqi Chen et al. "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems". In: *arXiv.org* (Dec. 2015), arXiv:1512.01274. arXiv: 1512.01274 [cs.DC] (cited on page 17).

[6] Caffe Community. *Caffe: Multi-GPU Usage*. https://github.com/BVLC/caffe/blob/master/docs/multigpu.md (cited on page 19).

[7] Yangqing Jia et al. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *Proceedings of the 22Nd ACM International Conference on Multimedia*. MM '14. Orlando, Florida, USA: ACM, 2014, pages 675–678. ISBN: 978-1-4503-3063-3. DOI: 10.1145/2647868.2654889. URL: http://doi.acm.org/10.1145/2647868.2654889 (cited on page 19).

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Edited by F. Pereira et al. Curran Associates, Inc., 2012, pages 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf (cited on pages 20, 21, 54).

[9] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pages 211–252. DOI: 10.1007/s11263-015-0816-y (cited on pages 21, 297).

[10] WordNet. *About WordNet.* http://wordnet.princeton.edu. 2010 (cited on pages 21, 31).

[11] Alisson Gray. *NVIDIA and IBM Cloud Support ImageNet Large Scale Visual Recognition Challenge*. https://devblogs.nvidia.com/parallelforall/nvidia-ibm-cloud-support-imagenet-large-scale-visual-recognition-challenge/ (cited on page 22).

[12] M. Everingham et al. “The Pascal Visual Object Classes (VOC) Challenge”. In: *International Journal of Computer Vision* 88.2 (June 2010), pages 303–338 (cited on page 22).

[13] AcademicTorrents Community and Curators. *ImageNet LSVRC 2015*. http://academictorrents.com/collection/imagenet-lsvrc-2015 (cited on page 24).

[14] Yangqing Jia and Evan Shelhamer. *Caffe Model Zoo: BAIR model license*. http://caffe.berkeleyvision.org/model_zoo.html#bvlc-model-license (cited on page 26).

[15] Tomasz Malisiewicz. *Deep Learning vs Big Data: Who owns what?* http://www.computervisionblog.com/2015/05/deep-learning-vs-big-data-who-owns-what.html (cited on page 27).

[16] Jeff Donahue. *BVLC Reference CaffeNet*. https://github.com/BVLC/caffe/tree/master/models/bvlc_reference_caffenet (cited on page 67).

[17] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *CoRR* abs/1409.1556 (2014). URL: http://arxiv.org/abs/1409.1556 (cited on pages 75, 76, 86, 183, 199, 254, 294).

[18] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS’10). Society for Artificial Intelligence and Statistics*. 2010 (cited on page 77).

[19] Dmytro Mishkin and Jiri Matas. “All you need is a good init”. In: *CoRR* abs/1511.06422 (2015). URL: http://arxiv.org/abs/1511.06422 (cited on pages 77, 86).

[20] Kaiming He et al. “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”. In: *CoRR* abs/1502.01852 (2015). URL: http://arxiv.org/abs/1502.01852 (cited on pages 77, 85).

[21] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *CoRR* abs/1512.03385 (2015). URL: http://arxiv.org/abs/1512.03385 (cited on pages 86, 90, 105, 107, 254, 294).

[22] Kaiming He et al. “Identity Mappings in Deep Residual Networks”. In: *CoRR* abs/1603.05027 (2016). URL: http://arxiv.org/abs/1603.05027 (cited on pages 86, 105, 254, 294).

[23] Christian Szegedy et al. “Going Deeper with Convolutions”. In: *Computer Vision and Pattern Recognition (CVPR)*. 2015. URL: http://arxiv.org/abs/1409.4842 (cited on pages 89, 90, 103).

[24] VLFeat Community. *VLFeat: Pre-trained Models*. http://www.vlfeat.org/matconvnet/pretrained/ (cited on pages 89, 99, 103, 104, 120).

[25] Jost Tobias Springenberg et al. “Striving for Simplicity: The All Convolutional Net”. In: *CoRR* abs/1412.6806 (2014). URL: http://arxiv.org/abs/1412.6806 (cited on pages 90, 109).

[26] ImageNet. *Large Scale Visual Recognition Challenge 2014 (ILSVRC2014)*. http://image-net.org/challenges/LSVRC/2014/results (cited on page 99).

[27] Kaiming He. *Deep Residual Networks*. https://github.com/KaimingHe/deep-residual-networks (cited on page 108).

[28] Wei Wu. *ResNet*. https://github.com/tornadomeet/ResNet (cited on page 108).

[29] Kaiming He. *ResNet: Should the convolution layers have biases?* https://github.com/KaimingHe/deep-residual-networks/issues/10#issuecomment-194037195 (cited on page 109).

[30] Forrest N. Iandola et al. "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size". In: *CoRR* abs/1602.07360 (2016). URL: http://arxiv.org/abs/1602.07360 (cited on pages 121, 123, 133, 294).

[31] Kaggle Team. *Challenges in Representation Learning: Facial Expression Recognition Challenge*. https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge (cited on pages 141, 159).

[32] Ian J. Goodfellow et al. "Challenges in Representation Learning: A Report on Three Machine Learning Contests". In: *Neural Information Processing: 20th International Conference, ICONIP 2013, Daegu, Korea, November 3-7, 2013. Proceedings, Part III*. Edited by Minho Lee et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pages 117–124. ISBN: 978-3-642-42051-1. DOI: 10.1007/978-3-642-42051-1_16. URL: https://doi.org/10.1007/978-3-642-42051-1_16 (cited on page 141).

[33] Jostine Ho. *Facial Emotion Recognition*. https://github.com/JostineHo/mememoji. 2016 (cited on pages 143, 156).

[34] Adrian Rosebrock. *Practical Python and OpenCV + Case Studies*. PyImageSearch.com, 2016. URL: https://www.pyimagesearch.com/practical-python-opencv/ (cited on pages 160, 162, 315).

[35] Adrian Rosebrock. *PyImageSearch Gurus*. https://www.pyimagesearch.com/pyimagesearch-gurus/. 2016 (cited on pages 160, 315).

[36] A. Quattoni and A. Torralba. "Recognizing indoor scenes". In: *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pages 413–420 (cited on pages 165, 166).

[37] Jonathan Krause et al. "3D Object Representations for Fine-Grained Categorization". In: *4th International IEEE Workshop on 3D Representation and Recognition (3dRR-13)*. Sydney, Australia, 2013 (cited on page 179).

[38] G. Levi and T. Hassncer. "Age and gender classification using convolutional neural networks". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. June 2015, pages 34–42. DOI: 10.1109/CVPRW.2015.7301352 (cited on page 203).

[39] Eran Eidinger, Roee Enbar, and Tal Hassner. "Age and Gender Estimation of Unfiltered Faces". In: *Trans. Info. For. Sec.* 9.12 (Dec. 2014), pages 2170–2179. ISSN: 1556-6013. DOI: 10.1109/TIFS.2014.2359646. URL: http://dx.doi.org/10.1109/TIFS.2014.2359646 (cited on page 204).

[40] Davis E. King. "Dlib-ml: A Machine Learning Toolkit". In: *J. Mach. Learn. Res.* 10 (Dec. 2009), pages 1755–1758. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1577069.1755843 (cited on pages 238, 299).

[41] Navneet Dalal and Bill Triggs. “Histograms of Oriented Gradients for Human Detection”. In: *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1 - Volume 01*. CVPR '05. Washington, DC, USA: IEEE Computer Society, 2005, pages 886–893. ISBN: 0-7695-2372-2. DOI: 10.1109/CVPR.2005.177. URL: http://dx.doi.org/10.1109/CVPR.2005.177 (cited on pages 238, 242, 247, 248).

[42] Pedro F. Felzenszwalb et al. “Object Detection with Discriminatively Trained Part-Based Models”. In: *IEEE Trans. Pattern Anal. Mach. Intell.* 32.9 (Sept. 2010), pages 1627–1645. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2009.167. URL: http://dx.doi.org/10.1109/TPAMI.2009.167 (cited on pages 238, 242).

[43] Ross B. Girshick et al. “Rich feature hierarchies for accurate object detection and semantic segmentation”. In: *CoRR* abs/1311.2524 (2013). URL: http://arxiv.org/abs/1311.2524 (cited on pages 247, 251, 260, 293).

[44] Paul Viola and Michael Jones. “Rapid object detection using a boosted cascade of simple features”. In: 2001, pages 511–518 (cited on page 247).

[45] TensorFlow Community. *Tensorflow Object Detection API*. https://github.com/tensorflow/models/tree/master/research/object_detection. 2017 (cited on pages 247, 263, 269, 285).

[46] Shaoqing Ren et al. “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”. In: *CoRR* abs/1506.01497 (2015). URL: http://arxiv.org/abs/1506.01497 (cited on pages 248, 253, 259, 260, 293).

[47] L. Fei-Fei, R. Fergus, and Pietro Perona. “Learning Generative Visual Models From Few Training Examples: An Incremental Bayesian Approach Tested on 101 Object Categories”. In: 2004 (cited on page 248).

[48] Joseph Redmon et al. “You Only Look Once: Unified, Real-Time Object Detection”. In: *CoRR* abs/1506.02640 (2015). URL: http://arxiv.org/abs/1506.02640 (cited on page 248).

[49] Joseph Redmon and Ali Farhadi. “YOLO9000: Better, Faster, Stronger”. In: *arXiv preprint arXiv:1612.08242* (2016) (cited on page 248).

[50] J.R.R. Uijlings et al. “Selective Search for Object Recognition”. In: *International Journal of Computer Vision* (2013). DOI: 10.1007/s11263-013-0620-5. URL: http://www.huppelen.nl/publications/selectiveSearchDraft.pdf (cited on page 251).

[51] Ross B. Girshick. “Fast R-CNN”. In: *CoRR* abs/1504.08083 (2015). arXiv: 1504.08083. URL: http://arxiv.org/abs/1504.08083 (cited on pages 252, 260, 293).

[52] Matthew D. Zeiler and Rob Fergus. “Visualizing and Understanding Convolutional Networks”. In: *CoRR* abs/1311.2901 (2013). URL: http://arxiv.org/abs/1311.2901 (cited on page 254).

[53] Andrew G. Howard et al. “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications”. In: *CoRR* abs/1704.04861 (2017). URL: http://arxiv.org/abs/1704.04861 (cited on pages 254, 294).

[54] Javier Rey. *Faster R-CNN: Down the rabbit hole of modern object detection*. https://tryolabs.com/blog/2018/01/18/faster-r-cnn-down-the-rabbit-hole-of-modern-object-detection/. 2018 (cited on pages 259, 260).

[55] Jonathan Huang et al. “Speed/accuracy trade-offs for modern convolutional object detectors”. In: *CoRR* abs/1611.10012 (2016). URL: http://arxiv.org/abs/1611.10012 (cited on pages 260, 263, 294).

[56] TryoLabs and Open Source Community. *luminoth - Deep Learning toolkit for Computer Vision*. https://github.com/tryolabs/luminoth. 2017 (cited on page 260).

[57] Andreas Mogelmose, Mohan Manubhai Trivedi, and Thomas B. Moeslund. "Vision-Based Traffic Sign Detection and Analysis for Intelligent Driver Assistance Systems: Perspectives and Survey". In: *Trans. Intell. Transport. Sys.* 13.4 (Dec. 2012), pages 1484–1497. ISSN: 1524-9050 (cited on page 261).

[58] M. Everingham et al. "The Pascal Visual Object Classes Challenge: A Retrospective". In: *International Journal of Computer Vision* 111.1 (Jan. 2015), pages 98–136 (cited on pages 267, 297).

[59] Tsung-Yi Lin et al. "Microsoft COCO: Common Objects in Context". In: *CoRR* abs/1405.0312 (2014). URL: http://arxiv.org/abs/1405.0312 (cited on pages 275, 297).

[60] O. M. Parkhi et al. "Cats and Dogs". In: *IEEE Conference on Computer Vision and Pattern Recognition*. 2012 (cited on page 277).

[61] Wei Liu et al. "SSD: Single Shot MultiBox Detector". In: *CoRR* abs/1512.02325 (2015). URL: http://arxiv.org/abs/1512.02325 (cited on pages 293–297).

[62] Christian Szegedy et al. "Scalable, High-Quality Object Detection". In: *CoRR* abs/1412.1441 (2014). URL: http://arxiv.org/abs/1412.1441 (cited on pages 293–295).

[63] Eddie Forson. *Understanding SSD MultiBox - Real-Time Object Detection in Deep Learning*. https://towardsdatascience.com/understanding-ssd-multibox-real-time-object-detection-in-deep-learning-495ef744fab. 2017 (cited on pages 296, 297).

[64] Davis E. King. "Max-Margin Object Detection". In: *CoRR* abs/1502.00046 (2015). URL: http://arxiv.org/abs/1502.00046 (cited on page 299).

[65] Davis King. *Dlib 18.6 released: Make your own object detector!* http://blog.dlib.net/2014/02/dlib-186-released-make-your-own-object.html. 2014 (cited on pages 299, 312).

[66] Davis King. *dlib - imglab*. https://github.com/davisking/dlib/tree/master/tools/imglab. 2016 (cited on pages 299, 312).

[67] Davis King. *Vehicle Detection with Dlib 19.5*. http://blog.dlib.net/2017/08/vehicle-detection-with-dlib-195_27.html. 2017 (cited on page 300).

[68] Leonard Richardson. *Beautiful Soup: We called him Tortoise because he taught us.* https://www.crummy.com/software/BeautifulSoup/. 2004 (cited on page 302).