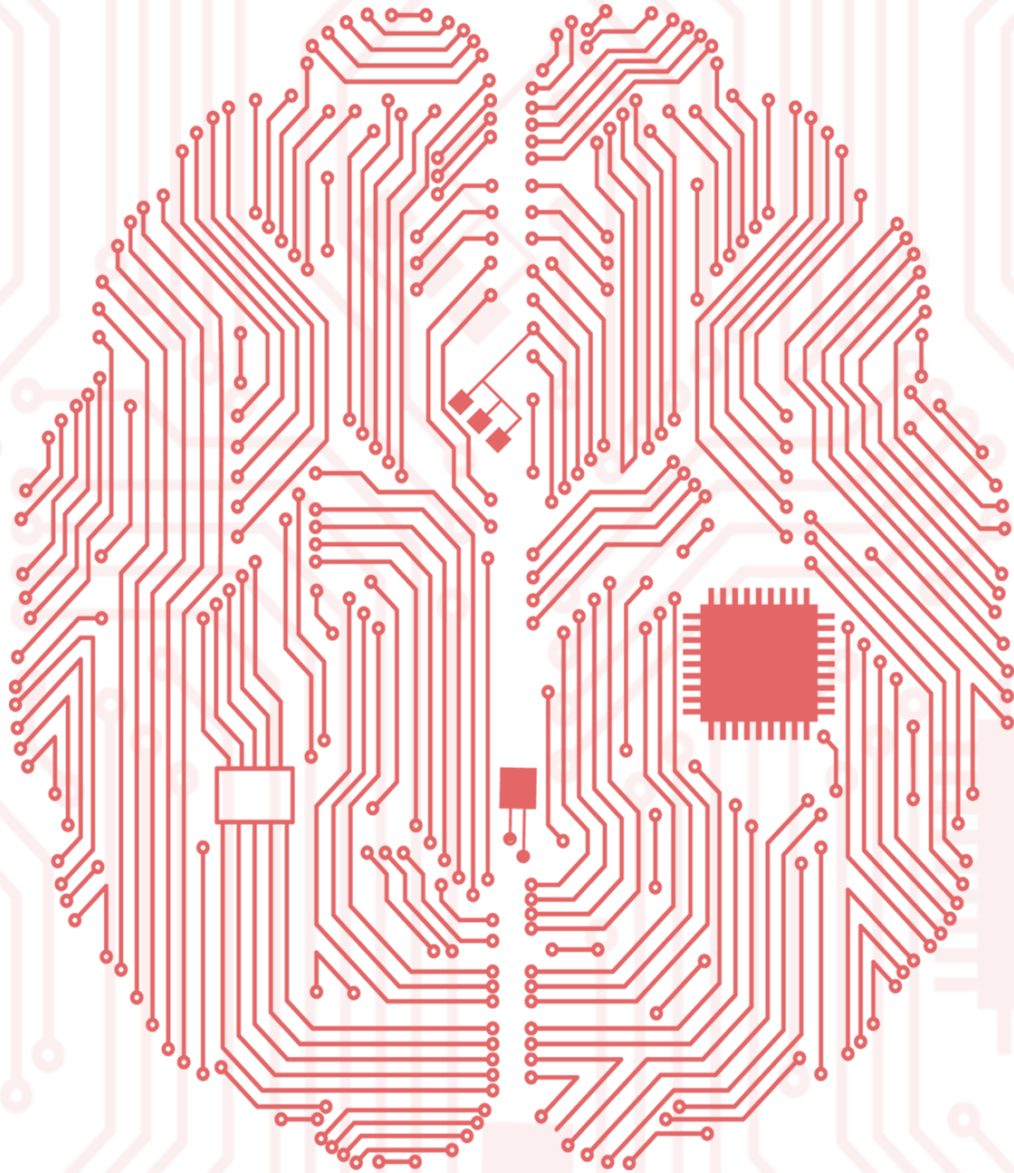


DEEP LEARNING FOR COMPUTER VISION



WITH PYTHON

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 recognize 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 of 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.

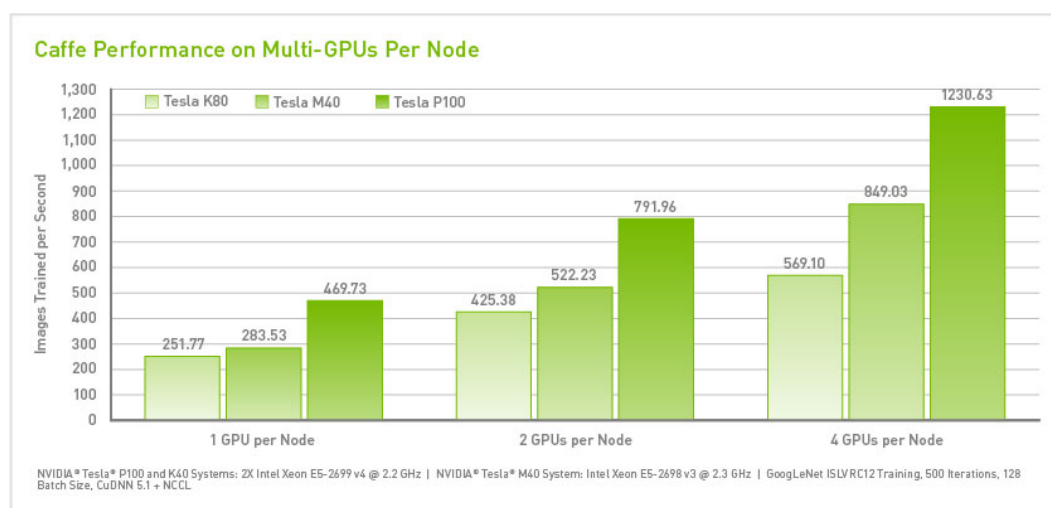


Figure 3.1: On the x -axis we have the number of GPUs (1-4) while the y -axis describes the number of images trained per second. When increasing from one to two GPUs we can inspect an increase of $\approx 1.82x$ performance. Jumping from one to four GPUs yields an increase of $\approx 2.71x$.

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.

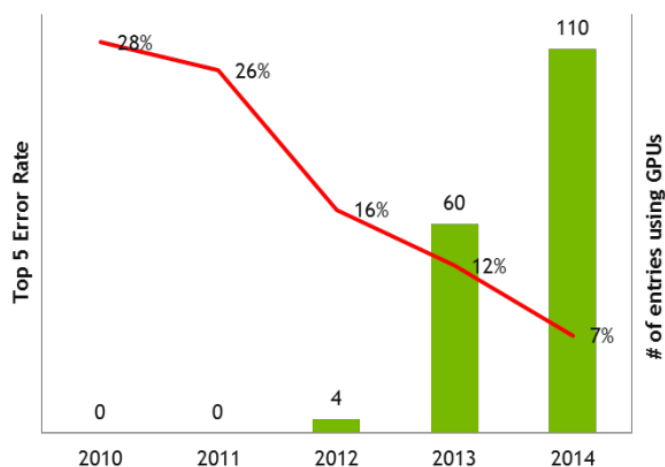


Figure 4.1: Since the seminal AlexNet architecture was introduced in 2012, Convolutional Neural Network methods have dominated the ILSVRC challenge, both in terms of accuracy and number of entries. (Image credit: NVIDIA [11])

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 ≈ 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/logm0>).

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/logm0>) 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: @umbc.edu
Full Name: Dr. Adrian Rosebrock
Organization: PylImageSearch

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 ≈ 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

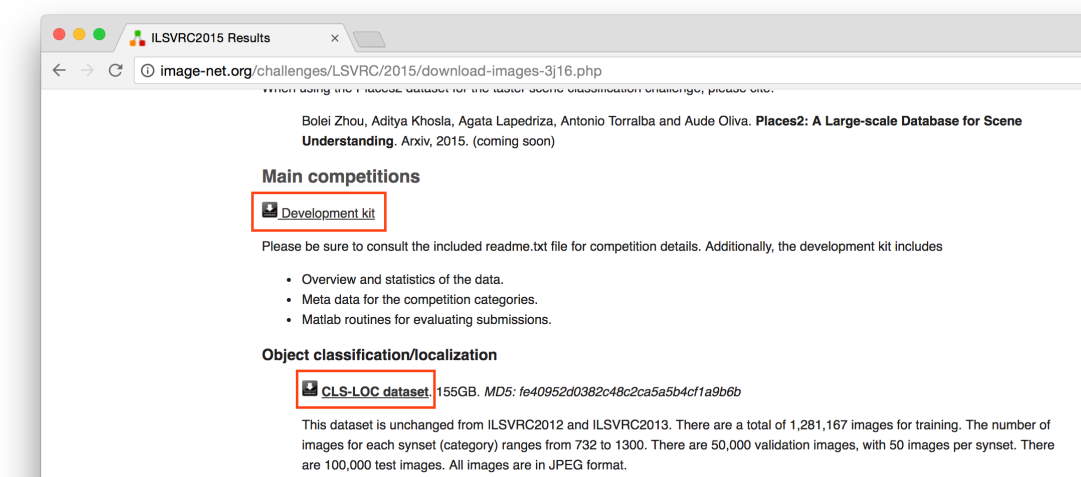


Figure 4.3: To download the entire ImageNet dataset you’ll need to download the development kit along with the large .tar archive of the ≈ 1.2 million images. (highlighted with red rectangles).

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* repackaging 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).

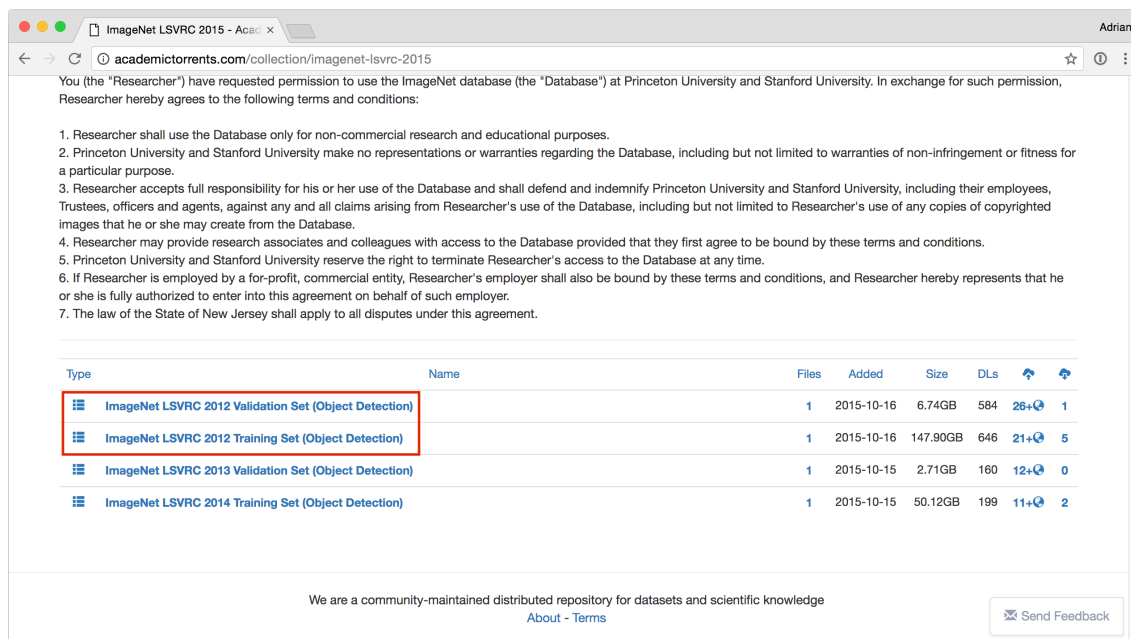


Figure 4.4: A screenshot of the AcademicTorrents website for ImageNet. Make sure you download the “ImageNet LSVRC 2012 Validation Set (Object Detection)” and “ImageNet LSVRC 2012 Training Set (Object Detection)” files, outlined in red.

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

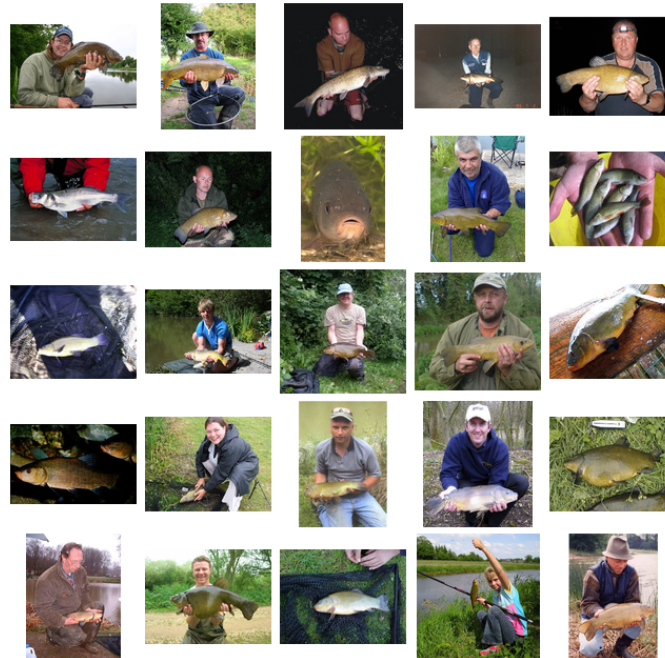


Figure 5.1: A sample of 25 images from the n01440764 syn-set. These images are examples of “tench”, a type of European freshwater fish.

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 file-names for the validation set. There are exactly 50,000 entries (one 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
| | |--- imagenet_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 “shortcut”) 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 ≈ 1.2 million (partial) image filenames for the training data:

```

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

```

Next, we need to define some validation configurations:

```

22 # define the paths to to the validation filenames along with the
23 # file that contains the ground-truth validation labels
24 VAL_LIST = path.sep.join([IMAGE_SETS_PATH, "val.txt"])
25 VAL_LABELS = path.sep.join([DEVKIT_PATH,
26     "ILSVRC2015_clsloc_validation_ground_truth.txt"])
27
28 # define the path to the validation files that are blacklisted
29 VAL_BLACKLIST = path.sep.join([DEVKIT_PATH,
30     "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:

```

32 # since we do not have access to the testing data we need to
33 # take a number of images from the training data and use it instead
34 NUM_CLASSES = 1000
35 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:

```

37 # define the path to the output training, validation, and testing
38 # lists
39 MX_OUTPUT = "/raid/datasets/imagenet"
40 TRAIN_MX_LIST = path.sep.join([MX_OUTPUT, "lists/train.lst"])
41 VAL_MX_LIST = path.sep.join([MX_OUTPUT, "lists/val.lst"])
42 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
|   |   |--- captcha_helper.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.buildBlacklist()

```

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 buildBlacklist(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 of 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 then 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-L0C/train/n02097130/n02097130_4602.JPEG
1 640 /raid/datasets/imagenet/ILSVRC2015/Data/CLS-L0C/train/n02276258/n02276258_7039.JPEG
2 375 /raid/datasets/imagenet/ILSVRC2015/Data/CLS-L0C/train/n03109150/n03109150_2152.JPEG
3 121 /raid/datasets/imagenet/ILSVRC2015/Data/CLS-L0C/train/n02483708/n02483708_3226.JPEG
4 977 /raid/datasets/imagenet/ILSVRC2015/Data/CLS-L0C/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: