Thank you for downloading the table of contents + sample chapters to *Deep Learning for Computer Vision with Python!*

This book has one goal — **to help *developers*, *researchers*, and *students* just like yourself become *experts* in deep learning for image classification and recognition.**

Whether this is the **first time you've worked with machine learning & neural networks** or **you're already a seasoned deep learning practitioner**, this book is engineered from the ground up to help you reach expert status.

Since this book covers a *huge* amount of content (**over 800+ pages**), I've decided to break the book down into ***three volumes*** called **"bundles".**

Each bundle *builds on top of the others* and includes *all chapters from the previous bundle.*

You should choose a bundle based on:

1. How **in-depth** you want to study deep learning and computer vision.
2. Your particular budget.

You can find a quick breakdown of the three bundles below:

- **Starter Bundle:** A great fit if you are taking your first step towards deep learning for image classification mastery.
- **Practitioner Bundle:** Perfect if you want to study deep learning *in-depth*, understand *advanced techniques*, and discover *common best practices and rules of thumb.*
- **ImageNet Bundle:** The *complete* deep learning for computer vision experience. Here I demonstrate how to train large-scale networks on the *massive* ImageNet dataset. ***You just can't beat this bundle.***

In the remainder of this PDF you'll find both the *table of contents* and *sample chapters* for each bundle:

- **Table of Contents**
  - **Starter Bundle:** Pages 4-10
  - **Practitioner Bundle:** Pages 11-15
  - **ImageNet Bundle:** Pages 16-20

- **Sample Chapters**
  - **Pages 21-31:** What is Deep Learning?
  - **Pages 32-45:** Training Your First CNN
  - **Pages 46-66:** Case Study: Breaking captchas with a CNN
  - **Pages 67-94:** Competing in Kaggle: Dogs vs. Cats
  - **Pages 95-117:** Training AlexNet on ImageNet

To see the full list of topics you'll master inside *Deep Learning for Computer Vision with Python*, just keep scrolling…

Contents

The *Practitioner Bundle* includes all chapters from the *Starter Bundle* plus…

Contents

The *ImageNet Bundle* includes all chapters from the *Starter Bundle* AND *Practitioner Bundle*.

I'll also be including a *special bonus guide* on Faster R-CNNs and SSDs for object detection in the *ImageNet Bundle* as well.

Contents

The following *"What is Deep Learning"* chapter is from the *Starter Bundle*...

2. What Is Deep Learning?

> *"Deep learning methods are representation-learning methods with multiple levels of representation, obtained by composing simple but nonlinear modules that each transform the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level. [...] The key aspect of deep learning is that these layers are not designed by human engineers: they are learned from data using a general-purpose learning procedure"* – Yann LeCun, Yoshua Bengio, and Geoffrey Hinton, Nature 2015. [9]

Deep learning is a subfield of machine learning, which is, in turn, a subfield of artificial intelligence (AI). For a graphical depiction of this relationship, please refer to Figure 2.1.

The central goal of AI is to provide a set of algorithms and techniques that can be used to solve problems that humans perform *intuitively* and *near automatically*, but are otherwise very challenging for computers. A great example of such a class of AI problems is interpreting and understanding the contents of an image – this task is something that a human can do with little-to-no effort, but it has proven to be *extremely difficult* for machines to accomplish.

While AI embodies a large, diverse set of work related to automatic machine reasoning (inference, planning, heuristics, etc.), the machine learning subfield tends to be *specifically interested* in **pattern recognition** and **learning from data**.

Artificial Neural Networks (ANNs) are a class of machine learning algorithms that learn from data and specialize in pattern recognition, inspired by the structure and function of the brain. As we'll find out, deep learning belongs to the family of ANN algorithms, and in most cases, the two terms can be used interchangeably. In fact, you may be surprised to learn that the deep learning field has been around for over *60 years*, going by different names and incarnations based on research trends, available hardware and datasets, and popular options of prominent researchers at the time.

In the remainder of this chapter, we'll review a brief history of deep learning, discuss what makes a neural network "deep", and discover the concept of "hierarchical learning" and how it has made deep learning one of the major success stories in modern day machine learning and computer vision.

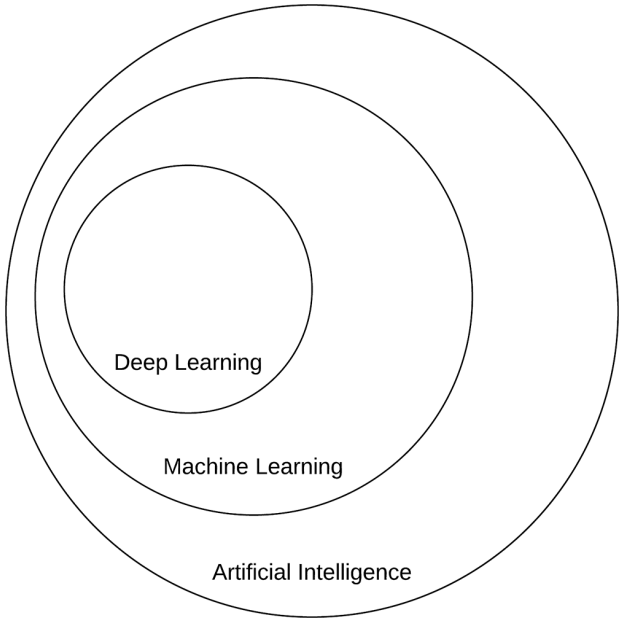

Figure 2.1: A Venn diagram describing deep learning as a subfield of machine learning which is in turn a subfield of artificial intelligence (Image inspired by Figure 1.4 of Goodfellow et al. [10]).

2.1 A Concise History of Neural Networks and Deep Learning

The history of neural networks and deep learning is a long, somewhat confusing one. It may surprise you to know that "deep learning" has existed since the 1940s undergoing various name changes, including *cybernetics*, *connectionism*, and the most familiar, *Artificial Neural Networks* (ANNs).

While *inspired* by the human brain and how its neurons interact with each other, ANNs are *not* meant to be realistic models of the brain. Instead, they are an inspiration, allowing us to draw parallels between a very basic model of the brain and how we can mimic some of this behavior through artificial neural networks. We'll discuss ANNs and the relation to the brain in Chapter 10.

The first neural network model came from McCulloch and Pitts in 1943 [11]. This network was a *binary classifier*, capable of recognizing two different categories based on some input. The problem was that the *weights* used to determine the class label for a given input needed to be *manually tuned* by a human – this type of model clearly does not scale well if a human operator is required to intervene.

Then, in the 1950s the seminal Perceptron algorithm was published by Rosenblatt [12, 13] – this model could *automatically* learn the weights required to classify an input (no human intervention required). An example of the Perceptron architecture can be seen in Figure 2.2. In fact, this automatic training procedure formed the basis of Stochastic Gradient Descent (SGD) which is still used to train *very deep* neural networks today.

During this time period, Perceptron-based techniques were all the rage in the neural network community. However, a 1969 publication by Minsky and Papert [14] effectively stagnated neural network research for nearly a decade. Their work demonstrated that a Perceptron with a linear activation function (regardless of depth) was merely a linear classifier, unable to solve nonlinear problems. The canonical example of a nonlinear problem is the XOR dataset in Figure 2.3. Take a second now to convince yourself that it is *impossible* to try a *single line* that can separate the blue

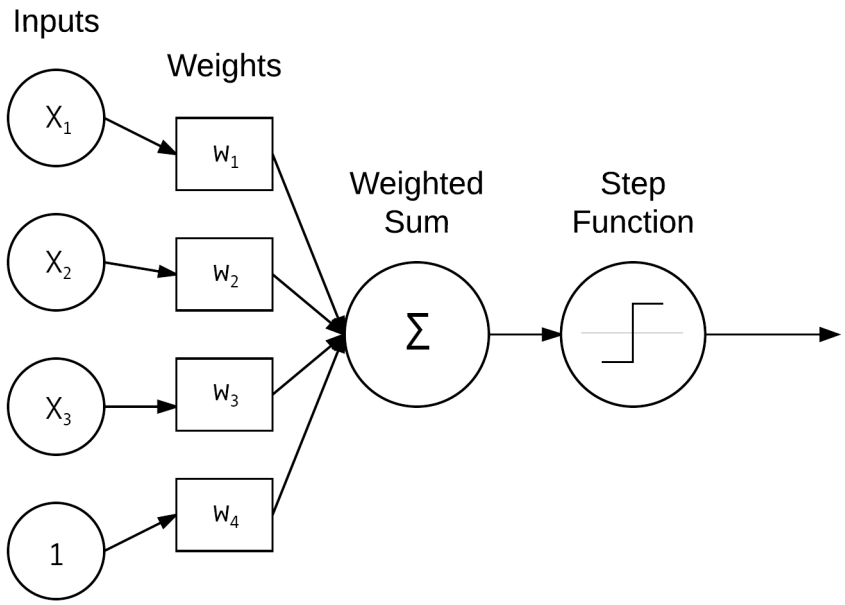

Figure 2.2: An example of the simple Perceptron network architecture that accepts a number of inputs, computes a weighted sum, and applies a step function to obtain the final prediction. We'll review the Perceptron in detail inside Chapter 10.

stars from the red circles.

Furthermore, the authors argued that (at the time) we did not have the computational resources required to construct large, deep neural networks (in hindsight, they were absolutely correct). This single paper alone *almost killed* neural network research.

Luckily, the backpropagation algorithm and the research by Werbos (1974) [15], Rumelhart (1986) [16], and LeCun (1998) [17] were able to resuscitate neural networks from what could have been an early demise. Their research in the backpropagation algorithm enabled *multi-layer feedforward* neural networks to be trained (Figure 2.4).

Combined with nonlinear activation functions, researchers could now learn nonlinear functions and solve the XOR problem, opening the gates to an entirely new area of research in neural networks. Further research demonstrated that neural networks are *universal approximators* [18], capable of approximating any continuous function (but placing no guarantee on whether or not the network can actually *learn* the parameters required to represent a function).

The backpropagation algorithm is the cornerstone of modern day neural networks allowing us to efficiently train neural networks and "teach" them to learn from their mistakes. But even so, at this time, due to (1) slow computers (compared to modern day machines) and (2) lack of large, labeled training sets, researchers were unable to (reliably) train neural networks that had more than two hidden layers – it was simply computationally infeasible.

Today, the latest incarnation of neural networks as we know it is called **deep learning**. What sets deep learning apart from its previous incarnations is that we have faster, specialized hardware with more available training data. We can now train networks with *many more hidden layers* that are capable of hierarchical learning where simple concepts are learned in the lower layers and more abstract patterns in the higher layers of the network.

Perhaps the quintessential example of applied deep learning to feature learning is the *Convo-*

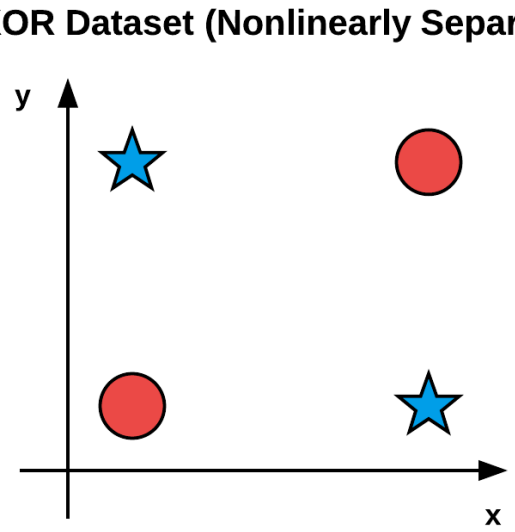

Figure 2.3: The XOR (E(X)clusive Or) dataset is an example of a nonlinear separable problem that the Perceptron *cannot* solve. Take a second to convince yourself that it is impossible to draw a single line that separates the blue stars from the red circles.

lutional Neural Network (LeCun 1988) [19] applied to handwritten character recognition which *automatically* learns discriminating patterns (called "filters") from images by sequentially stacking layers on top of each other. Filters in lower levels of the network represent edges and corners, while higher level layers use the edges and corners to learn more abstract concepts useful for discriminating between image classes.

In many applications, CNNs are now considered the most powerful image classifier and are currently responsible for pushing the state-of-the-art forward in computer vision subfields that leverage machine learning. For a more thorough review of the history of neural networks and deep learning, please refer to Goodfellow et al. [10] as well as this excellent blog post by Jason Brownlee at Machine Learning Mastery [20].

2.2 Hierarchical Feature Learning

Machine learning algorithms (generally) fall into three camps – *supervised*, *unsupervised*, and *semi-supervised* learning. We'll discuss supervised and unsupervised learning in this chapter while saving semi-supervised learning for a future discussion.

In the supervised case, a machine learning algorithm is given both a set of *inputs* and *target outputs*. The algorithm then tries to learn patterns that can be used to automatically map input data points to their correct target output. Supervised learning is similar to having a teacher watching you take a test. Given your previous knowledge, you do your best to mark the correct answer on your exam; however, if you are incorrect, your teacher guides you toward a better, more educated guess the next time.

In an unsupervised case, machine learning algorithms try to automatically discover discriminating features *without* any hints as to what the inputs are. In this scenario, our student tries to group similar questions and answers together, even though the student does not know what the correct answer is *and* the teacher is not there to provide them with the true answer. Unsupervised

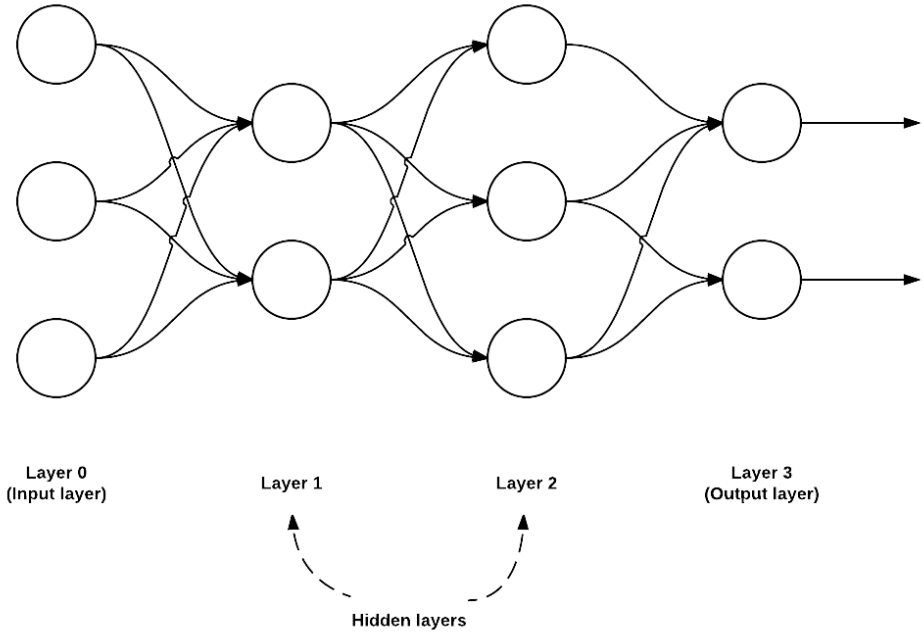

Figure 2.4: A multi-layer, feedforward network architecture with an input layer (3 nodes), two hidden layers (2 nodes in the first layer and 3 nodes in the second layer), and an output layer (2 nodes).

learning is clearly a more challenging problem than supervised learning – by knowing the answers (i.e., target outputs), we can more easily define discriminate patterns that can map input data to the correct target classification.

In the context of machine learning applied to image classification, the goal of a machine learning algorithm is to take these sets of images and identify patterns that can be used to discriminate various image classes/objects from one another.

In the past, we used *hand-engineered features* to quantify the contents of an image – we *rarely* used raw pixel intensities as inputs to our machine learning models, as is now common with deep learning. For each image in our dataset, we performed *feature extraction*, or the process of taking an input image, quantifying it according to some algorithm (called a *feature extractor* or *image descriptor*), and returning a vector (i.e., a list of numbers) that aimed to quantify the contents of an image. Figure 2.5 below depicts the process of quantifying an image containing prescription pill medication via a series of blackbox color, texture, and shape image descriptors.

Our hand-engineered features attempted to encode texture (Local Binary Patterns [21], Haralick texture [22]), shape (Hu Moments [23], Zernike Moments [24]), and color (color moments, color histograms, color correlograms [25]).

Other methods such as keypoint detectors (FAST [26], Harris [27], DoG [28], to name a few) and local invariant descriptors (SIFT [28], SURF [29], BRIEF [30], ORB [31], etc.) describe *salient* (i.e., the most "interesting") regions of an image.

Other methods such as Histogram of Oriented Gradients (HOG) [32] proved to be very good at detecting objects in images when the viewpoint angle of our image did not vary dramatically from what our classifier was trained on. An example of using the HOG + Linear SVM detector method

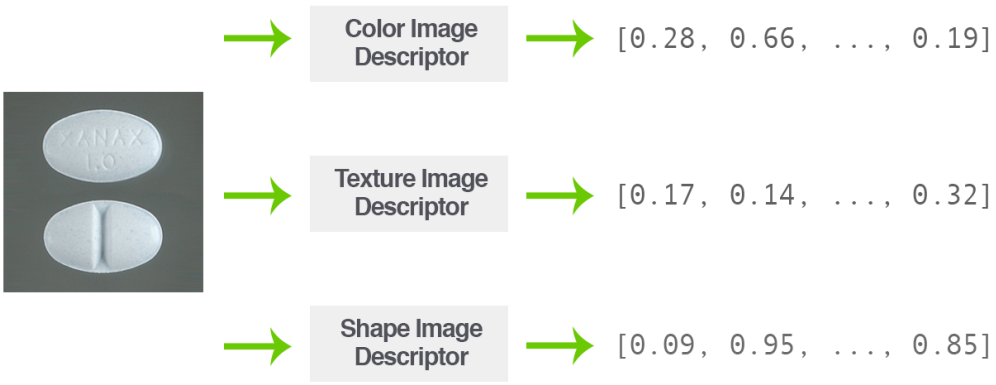

Figure 2.5: Quantifying the contents of an image containing a prescription pill medication via a series of blackbox color, texture, and shape image descriptors.

can be seen in Figure 2.6 where we detect the presence of stop signs in images.

For a while, research in object detection in images was guided by HOG and its variants, including computationally expensive methods such as the Deformable Parts Model [34] and Exemplar SVMs [35].

R For a more in-depth study of image descriptors, feature extraction, and the process it plays in computer vision, be sure to refer to the PyImageSearch Gurus course [33].

In each of these situations, an algorithm was *hand-defined* to quantify and encode a particular aspect of an image (i.e., shape, texture, color, etc.). Given an input image of pixels, we would apply our hand-defined algorithm to the pixels, and in return receive a feature vector quantifying the image contents – the image pixels themselves did not serve a purpose other than being inputs to our feature extraction process. The feature vectors that resulted from feature extraction were what we were truly interested in as they served as inputs to our machine learning models.

Deep learning, and specifically Convolutional Neural Networks, take a different approach. Instead of hand-defining a set of rules and algorithms to extract features from an image, **these features are instead automatically learned from the training process**.

Again, let's return to the goal of machine learning: *computers should be able to learn from experience (i.e., examples) of the problem they are trying to solve*.

Using deep learning, we try to understand the problem in terms of a hierarchy of concepts. Each concept builds on top of the others. Concepts in the lower level layers of the network encode some basic representation of the problem, whereas higher level layers *use these basic layers* to form more abstract concepts. This hierarchical learning allows us to *completely remove* the hand-designed feature extraction process and treat CNNs as end-to-end learners.

Given an image, we supply the pixel intensity values as **inputs** to the CNN. A series of **hidden layers** are used to extract features from our input image. These hidden layers build upon each other in a hierarchal fashion. At first, only edge-like regions are detected in the lower level layers of the network. These edge regions are used to define corners (where edges intersect) and contours (outlines of objects). Combining corners and contours can lead to abstract "object parts" in the next layer.

Again, keep in mind that the types of concepts these filters are learning to detect are *automatically learned* – there is no intervention by us in the learning process. Finally, **output** layer is

Figure 2.6: The HOG + Linear SVM object detection framework applied to detecting the location of stop signs in images, as covered inside the PyImageSearch Gurus course [33].

used to classify the image and obtain the output class label – the output layer is either *directly* or *indirectly* influenced by every other node in the network.

We can view this process as hierarchical learning: each layer in the network uses the output of previous layers as "building blocks" to construct increasingly more abstract concepts. These layers are learned *automatically* – there is *no hand-crafted feature engineering* taking place in our network. Figure 2.7 compares classic image classification algorithms using hand-crafted features to representation learning via deep learning and Convolutional Neural Networks.

One of the primary benefits of deep learning and Convolutional Neural Networks is that it allows us to skip the feature extraction step and instead focus on process of training our network to learn these filters. However, as we'll find out later in this book, training a network to obtain reasonable accuracy on a given image dataset isn't always an easy task.

2.3 How "Deep" Is Deep?

To quote Jeff Dean from his 2016 talk, *Deep Learning for Building Intelligent Computer Systems* [36]:

> *"When you hear the term deep learning, just think of a large, deep neural net. Deep refers to the number of layers typically and so this kind of the popular term that's been adopted in the press."*

This is an excellent quote as it allows us to conceptualize deep learning as large neural networks where layers build on top of each other, gradually increasing in depth. **The problem is we still don't have a concrete answer to the question, "How many layers does a neural network need to be considered *deep?*"**

The short answer is there is ***no consensus*** amongst experts on the depth of a network to be considered deep [10].

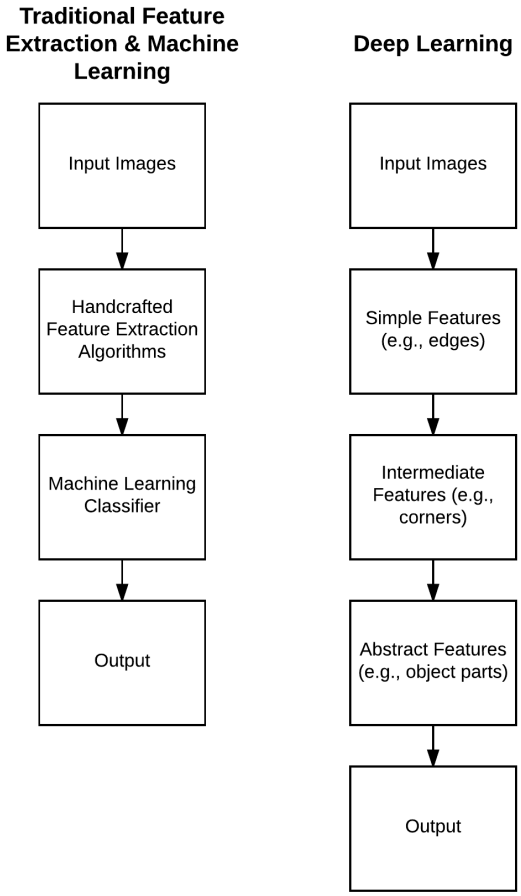

Figure 2.7: **Left:** Traditional process of taking an input set of images, applying hand-designed feature extraction algorithms, followed by training a machine learning classifier on the features. **Right:** Deep learning approach of stacking layers on top of each other that *automatically* learn more complex, abstract, and discriminating features.

And now we need to look at the question of network type. By definition, a Convolutional Neural Network (CNN) is a type of deep learning algorithm. But suppose we had a CNN with only one convolutional layer – is a network that is shallow, but yet still belongs to a family of algorithms inside the deep learning camp considered to be "deep"?

My personal opinion is that any network with greater than two hidden layers can be considered "deep". My reasoning is based on previous research in ANNs that were heavily handicapped by:

1. Our lack of large, labeled datasets available for training
2. Our computers being too slow to train large neural networks
3. Inadequate activation functions

Because of these problems, we could not easily train networks with more than two hidden layers during the 1980s and 1990s (and prior, of course). In fact, Geoff Hinton supports this sentiment in his 2016 talk, *Deep Learning* [37], where he discussed why the previous incarnations of deep learning (ANNs) did not take off during the 1990s phase:

1. Our labeled datasets were thousands of times too small.

2. Our computers were millions of times too slow.
3. We initialized the network weights in a stupid way.
4. We used the wrong type of nonlinearity activation function.

All of these reasons point to the fact that training networks with a depth larger than two hidden layers were a futile, if not a computational, impossibility.

In the current incarnation we can see that the tides have changed. We now have:

1. Faster computers
2. Highly optimized hardware (i.e., GPUs)
3. Large, labeled datasets in the order of millions of images
4. A better understanding of weight initialization functions and what does/does not work
5. Superior activation functions and an understanding regarding why previous nonlinearity functions stagnated research

Paraphrasing Andrew Ng from his 2013 talk, *Deep Learning, Self-Taught Learning and Unsupervised Feature Learning* [38], we are now able to construct deeper neural networks and train them with more data.

As the *depth* of the network increases, so does the *classification accuracy*. This behavior is different from traditional machine learning algorithms (i.e., logistic regression, SVMs, decision trees, etc.) where we reach a plateau in performance even as available training data increases. A plot inspired by Andrew Ng's 2015 talk, *What data scientists should know about deep learning*, [39] can be seen in Figure 2.8, providing an example of this behavior.

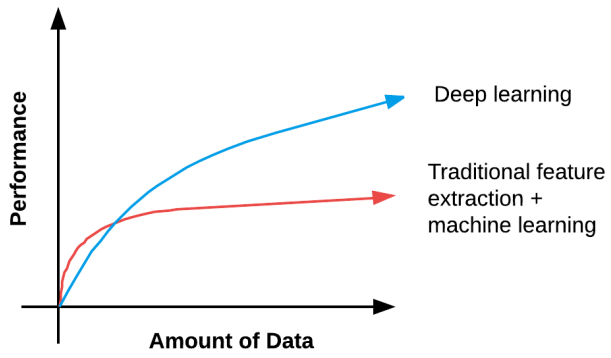

Figure 2.8: As the amount of data available to deep learning algorithms increases, accuracy does as well, substantially outperforming traditional feature extraction + machine learning approaches.

As the amount of training data increases, our neural network algorithms obtain higher classification accuracy, whereas previous methods plateau at a certain point. Because of the relationship between higher accuracy and more data, we tend to associate *deep learning* with *large datasets* as well.

When working on your own deep learning applications, I suggest using the following rule of thumb to determine if your given neural network is deep:

1. Are you using a *specialized* network architecture such as Convolutional Neural Networks, Recurrent Neural Networks, or Long Short-Term Memory (LSTM) networks? If so, **yes, you are performing deep learning**.
2. Does your network have a depth > 2? If yes, **you are doing deep learning**.
3. Does your network have a depth > 10? If so, **you are performing *very* deep learning** [40].

All that said, try not to get caught up in the buzzwords surrounding deep learning and what is/is not deep learning. At the very core, deep learning has gone through a number of different

incarnations over the past 60 years based on various schools of thought – **but *each* of these schools of thought centralize around artificial neural networks inspired by the structure and function of the brain**. Regardless of network depth, width, or specialized network architecture, you're *still* performing machine learning using artificial neural networks.

2.4 Summary

This chapter addressed the complicated question of *"What is deep learning?"*.

As we found out, deep learning has been around since the 1940s, going by different names and incarnations based on various schools of thought and popular research trends at a given time. At the very core, deep learning belongs to the family of Artificial Neural Networks (ANNs), a set of algorithms that learn patterns inspired by the structure and function of the brain.

There is no consensus amongst experts on exactly what makes a neural network "deep"; however, we know that:

1. Deep learning algorithms learn in a hierarchical fashion and therefore stack multiple layers on top of each other to learn increasingly more abstract concepts.
2. A network should have > 2 layers to be considered "deep" (this is my anecdotal opinion based on decades of neural network research).
3. A network with > 10 layers is considered *very deep* (although this number will change as architectures such as ResNet have been successfully trained with over 100 layers).

If you feel a bit confused or even overwhelmed after reading this chapter, don't worry – the purpose here was simply to provide an extremely high-level overview of deep learning and what exactly "deep" means.

This chapter also introduced a number of concepts and terms you may be unfamiliar with, including pixels, edges, and corners – our next chapter will address these types of image basics and give you a concrete foundation to stand on. We'll then start to move into the fundamentals of neural networks, allowing us to graduate to deep learning and Convolutional Neural Networks later in this book. While this chapter was admittedly high-level, the rest of the chapters of this book will be extremely hands-on, allowing you to master deep learning for computer vision concepts.

This *"Training Your First CNN"* chapter is from the *Starter Bundle…*

12. Training Your First CNN

Now that we've reviewed the fundamentals of Convolutional Neural Networks, we are ready to implement our first CNN using Python and Keras. We'll start the chapter with a quick review of Keras configurations you should keep in mind when constructing and training your own CNNs.

We'll then implement ShallowNet, which as the name suggests, is a very shallow CNN with only a single `CONV` layer. However, don't let the simplicity of this network fool you – as our results will demonstrate, ShallowNet is capable of obtaining higher classification accuracy on both CIFAR-10 and the Animals dataset than *any other method* we've reviewed thus far in this book.

12.1 Keras Configurations and Converting Images to Arrays

Before we can implement `ShallowNet`, we first need to review the `keras.json` configuration file and how the settings inside this file will influence how you implement your own CNNs. We'll also implement a second image preprocessor called `ImageToArrayPreprocessor` which accepts an input image and then converts it to a NumPy array that Keras can work with.

12.1.1 Understanding the keras.json Configuration File

The first time you import the Keras library into your Python shell/execute a Python script that imports Keras, behind the scenes Keras generates a `keras.json` file in your home directory. You can find this configuration file in `~/.keras/keras.json`.

Go ahead and open the file up now and take a look at its contents:

```
{
    "epsilon": 1e-07,
    "floatx": "float32",
    "image_data_format": "channels_last",
    "backend": "tensorflow"
}
```

You'll notice that this JSON-encoded dictionary has four keys and four corresponding values. The `epsilon` value is used in a variety of locations throughout the Keras library to prevent division by zero errors. The default value of `1e-07` is suitable and should not be changed. We then have the `floatx` value which defines the floating point precision – it is safe to leave this value at `float32`.

The final two configurations, `image_data_format` and `backend`, are *extremely important*. By default, the Keras library uses the *TensorFlow* numerical computation backend. We can also use the *Theano* backend simply by replacing `tensorflow` with `theano`.

You'll want to keep these backends in mind when *developing* your own deep learning networks and when you *deploy* them to other machines. Keras does a fantastic job abstracting the backend, allowing you to write deep learning code that is compatible with *either* backend (and surely more backends to come in the future), and for the most part, you'll find that both computational backends will give you the same result. If you find your results are inconsistent or your code is returning strange errors, check your backend first and make sure the setting is what you expect it to be.

Finally, we have the `image_data_format` which can accept two values: `channels_last` or `channels_first`. As we know from previous chapters in this book, images loaded via OpenCV are represented in `(rows, columns, channels)` ordering, which is what Keras calls `channels_last`, as the channels are the last dimension in the array.

Alternatively, we can set `image_data_format` to be `channels_first` where our input images are represented as `(channels, rows, columns)` – notice how the number of channels is the first dimension in the array.

Why the two settings? In the Theano community, users tended to use *channels first* ordering. However, when TensorFlow was released, their tutorials and examples used *channels last* ordering. This discrepancy caused a bit of a problem when using Keras as code compatible with Theano because it may not be compatible with TensorFlow depending on how the programmer built their network. Thus, Keras introduced a special function called `img_to_array` which accepts an input image and then orders the channels correctly based on the `image_data_format` setting.

In general, you can leave the `image_data_format` setting as `channels_last` and Keras will take care of the dimension ordering for you regardless of backend; however, I do want to call this situation to your attention just in case you are working with legacy Keras code and notice that a different image channel ordering is used.

12.1.2 The Image to Array Preprocessor

As I mentioned above, the Keras library provides the `img_to_array` function that accepts an input image and then properly orders the channels based on our `image_data_format` setting. We are going to wrap this function inside a new class named `ImageToArrayPreprocessor`. Creating a class with a special `preprocess` function, just like we did in Chapter 7 when creating the `SimplePreprocessor` to resize images, will allow us to create "chains" of preprocessors to efficiently prepare images for training and testing.

To create our image-to-array preprocessor, create a new file named `imagetoarraypreprocessor.py` inside the `preprocessing` sub-module of `pyimagesearch`:

```
|--- pyimagesearch
|    |--- __init__.py
|    |--- datasets
|    |    |--- __init__.py
|    |    |--- simpledatasetloader.py
|    |--- preprocessing
|    |    |--- __init__.py
|    |    |--- imagetoarraypreprocessor.py
|    |    |--- simplepreprocessor.py
```

From there, open the file and insert the following code:

```
1  # import the necessary packages
2  from keras.preprocessing.image import img_to_array
3
4  class ImageToArrayPreprocessor:
5      def __init__(self, dataFormat=None):
6          # store the image data format
7          self.dataFormat = dataFormat
8
9      def preprocess(self, image):
10         # apply the Keras utility function that correctly rearranges
11         # the dimensions of the image
12         return img_to_array(image, data_format=self.dataFormat)
```

Line 2 imports the `img_to_array` function from Keras.

We then define the constructor to our `ImageToArrayPreprocessor` class on **Lines 5-7**. The constructor accepts an optional parameter named `dataFormat`. This value defaults to `None`, which indicates that the setting inside `keras.json` should be used. We could also explicitly supply a `channels_first` or `channels_last` string, but it's best to let Keras choose which image dimension ordering to used based on the configuration file.

Finally, we have the `preprocess` function on **Lines 9-12**. This method:

1. Accepts an `image` as input.
2. Calls `img_to_array` on the `image`, ordering the channels based on our configuration file/the value of `dataFormat`.
3. Returns a new NumPy array with the channels properly ordered.

The benefit of defining a *class* to handle this type of image preprocessing rather than simply calling `img_to_array` on every single image is that we can now *chain* preprocessors together as we load datasets from disk.

For example, let's suppose we wished to resize all input images to a fixed size of 32×32 pixels. To accomplish this, we would need to initialize our `SimpleProcessor` from Chapter 7:

```
sp = SimplePreprocessor(32, 32)
```

After the image is resized, we then need to apply the properly channel ordering – this can be accomplished using our `ImageToArrayPreprocessor` above:

```
iap = ImageToArrayPreprocessor()
```

Now, suppose we wished to load an image dataset from disk and prepare all images in the dataset for training. Using the `SimpleDatasetLoader` from Chapter 7, our task becomes very easy:

```
sdl = SimpleDatasetLoader(preprocessors=[sp, iap])
(data, labels) = sdl.load(imagePaths, verbose=500)
```

Notice how our image preprocessors are *chained* together and will be applied in *sequential order*. For every image in our dataset, we'll first apply the `SimplePreprocessor` to resize it to

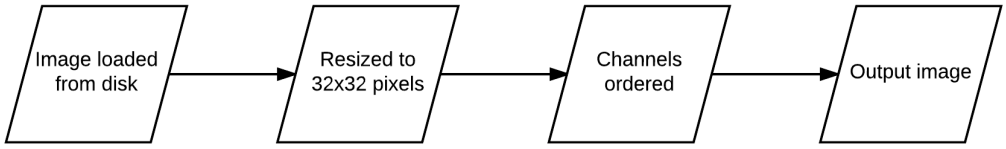

Figure 12.1: An example image pre-processing pipeline that (1) loads an image from disk, (2) resizes it to 32×32 pixels, (3) orders the channel dimensions, and (4) outputs the image.

32×32 pixels. Once the image is resized, the `ImageToArrayPreprocessor` is applied to handle ordering the channels of the image. This image processing pipeline can be visualized in Figure 12.1.

Chaining simple preprocessors together in this manner, where each preprocessor is responsible for *one, small job*, is an easy way to build an extendable deep learning library dedicated to classifying images. We'll make use of these preprocessors in the next section as well as define more advanced preprocessors in both the *Practitioner Bundle* and *ImageNet Bundle*.

12.2 ShallowNet

Inside this section, we'll implement the ShallowNet architecture. As the name suggests, the ShallowNet architecture contains only a few layers – the entire network architecture can be summarized as: `INPUT => CONV => RELU => FC`

This simple network architecture will allow us to get our feet wet implementing Convolutional Neural Networks using the Keras library. After implementing ShallowNet, I'll apply it to the Animals and CIFAR-10 datasets. As our results will demonstrate, CNNs are able to *dramatically outperform* the previous image classification methods discussed in this book.

12.2.1 Implementing ShallowNet

To keep our `pyimagesearch` package tidy, let's create a new sub-module inside `nn` named `conv` where all our CNN implementations will live:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- datasets
|    |--- nn
|    |    |--- __init__.py
...
|    |    |--- conv
|    |    |    |--- __init__.py
|    |    |    |--- shallownet.py
|    |--- preprocessing
```

Inside the `conv` sub-module, create a new file named `shallownet.py` to store our ShallowNet architecture implementation. From there, open up the file and insert the following code:

```
1 # import the necessary packages
2 from keras.models import Sequential
3 from keras.layers.convolutional import Conv2D
```

```
4 from keras.layers.core import Activation
5 from keras.layers.core import Flatten
6 from keras.layers.core import Dense
7 from keras import backend as K
```

Lines 2-7 import our required Python packages. The `Conv2D` class is the Keras implementation of the convolutional layer discussed in Section 11.1. We then have the `Activation` class, which as the name suggests, handles applying an activation function to an input. The `Flatten` classes takes our multi-dimensional volume and "flattens" it into a 1D array prior to feeding the inputs into the `Dense` (i.e, fully-connected) layers.

When implementing network architectures, I prefer to define them inside a class to keep the code organized – we'll do the same here:

```
9  class ShallowNet:
10     @staticmethod
11     def build(width, height, depth, classes):
12         # initialize the model along with the input shape to be
13         # "channels last"
14         model = Sequential()
15         inputShape = (height, width, depth)
16
17         # if we are using "channels first", update the input shape
18         if K.image_data_format() == "channels_first":
19             inputShape = (depth, height, width)
```

On **Line 9** we define the `ShallowNet` class and then define a `build` method on **Line 11**. Every CNN that we implement inside this book will have a `build` method – this function will accept a number of parameters, construct the network architecture, and then return it to the calling function. In this case, our `build` method requires four parameters:

- `width`: The width of the input images that will be used to train the network (i.e., number of columns in the matrix).
- `height`: The height of our input images (i.e., the number of rows in the matrix)
- `depth`: The number of channels in the input image.
- `classes`: The total number of classes that our network should learn to predict. For Animals, `classes=3` and for CIFAR-10, `classes=10`.

We then initialize the `inputShape` to the network on **Line 15** assuming "channels last" ordering. **Line 18 and 19** make a check to see if the Keras backend is set to "channels first", and if so, we update the `inputShape`. It's common practice to include **Lines 15-19** for nearly every CNN that you build, thereby ensuring that your network will work regardless of how a user is ordering the channels of their image.

Now that our `inputShape` is defined, we can start to build the ShallowNet architecture:

```
21         # define the first (and only) CONV => RELU layer
22         model.add(Conv2D(32, (3, 3), padding="same",
23             input_shape=inputShape))
24         model.add(Activation("relu"))
```

On **Line 24** we define the first (and only) convolutional layer. This layer will have 32 filters (K) each of which are 3×3 (i.e., square $F \times F$ filters). We'll apply `same` padding to ensure the size of output of the convolution operation matches the input (using `same` padding isn't strictly necessary

for this example, but it's a good habit to start forming now). After the convolution we apply an ReLU activation on **Line 24**.

Let's finish building ShallowNet:

```
26          # softmax classifier
27          model.add(Flatten())
28          model.add(Dense(classes))
29          model.add(Activation("softmax"))
30
31          # return the constructed network architecture
32          return model
```

In order to apply our fully-connected layer, we first need to flatten the multi-dimensional representation into a 1D list. The flattening operation is handled by the `Flatten` call on **Line 27**. Then, a `Dense` layer is created using the same number of nodes as our output class labels (**Line 28**). **Line 29** applies a softmax activation function which will give us the class label probabilities for each class. The ShallowNet architecture is returned to the calling function on **Line 32**.

Now that ShallowNet has been defined, we can move on to creating the actual "driver scripts" used to load a dataset, preprocess it, and then train the network. We'll look at two examples that leverage ShallowNet – Animals and CIFAR-10.

12.2.2 ShallowNet on Animals

To train ShallowNet on the Animals dataset, we need to create a separate Python file. Open up your favorite IDE, create a new file named `shallownet_animals.py`, ensuring that it is in the same directory level as our `pyimagesearch` module (or you have added `pyimagesearch` to the list of paths your Python interpreter/IDE will check when running a script).

From there, we can get to work:

```
1  # import the necessary packages
2  from sklearn.preprocessing import LabelBinarizer
3  from sklearn.model_selection import train_test_split
4  from sklearn.metrics import classification_report
5  from pyimagesearch.preprocessing import ImageToArrayPreprocessor
6  from pyimagesearch.preprocessing import SimplePreprocessor
7  from pyimagesearch.datasets import SimpleDatasetLoader
8  from pyimagesearch.nn.conv import ShallowNet
9  from keras.optimizers import SGD
10 from imutils import paths
11 import matplotlib.pyplot as plt
12 import numpy as np
13 import argparse
```

Lines 2-13 import our required Python packages. Most of these imports you've seen from previous examples, but I do want to call your attention to **Lines 5-7** where we import our `ImageToArrayPreprocessor`, `SimplePreprocessor`, and `SimpleDatasetLoader` – these classes will form the actual *pipeline* used to process images before passing them through our network. We then import `ShallowNet` on **Line 8** along with `SGD` on **Line 9** – we'll be using Stochastic Gradient Descent to train ShallowNet.

Next, we need to parse our command line arguments and grab our image paths:

```
15 # construct the argument parser and parse the arguments
16 ap = argparse.ArgumentParser()
17 ap.add_argument("-d", "--dataset", required=True,
18      help="path to input dataset")
19 args = vars(ap.parse_args())
20
21 # grab the list of images that we'll be describing
22 print("[INFO] loading images...")
23 imagePaths = list(paths.list_images(args["dataset"]))
```

Our script requires only a single switch here, `--dataset`, which is the path to the directory containing our Animals dataset. **Line 23** then grabs the file paths to all 3,000 images inside Animals.

Remember how I was talking about creating a pipeline to load and process our dataset? Let's see how that is done now:

```
25 # initialize the image preprocessors
26 sp = SimplePreprocessor(32, 32)
27 iap = ImageToArrayPreprocessor()
28
29 # load the dataset from disk then scale the raw pixel intensities
30 # to the range [0, 1]
31 sdl = SimpleDatasetLoader(preprocessors=[sp, iap])
32 (data, labels) = sdl.load(imagePaths, verbose=500)
33 data = data.astype("float") / 255.0
```

Line 26 defines the `SimpleProcessor` used to resize input images to 32×32 pixels. The `ImageToArrayPreprocessor` is then instantiated on **Line 27** to handle channel ordering.

We combine these preprocessors together on **Line 31** where we initialize the `SimpleDatasetLoader`. Take a look at the `preprocessors` parameter of the constructor – we are supplying a *list* of preprocessors that will be applied in *sequential order*. First, a given input image will be resized to 32×32 pixels. Then, the resized image will be have its channels ordered according to our `keras.json` configuration file. **Line 32** loads the images (applying the preprocessors) and the class labels. We then scale the images to the range $[0, 1]$.

Now that the data and labels are loaded, we can perform our training and testing split, along with one-hot encoding the labels:

```
35 # partition the data into training and testing splits using 75% of
36 # the data for training and the remaining 25% for testing
37 (trainX, testX, trainY, testY) = train_test_split(data, labels,
38      test_size=0.25, random_state=42)
39
40 # convert the labels from integers to vectors
41 trainY = LabelBinarizer().fit_transform(trainY)
42 testY = LabelBinarizer().fit_transform(testY)
```

Here we are using 75% of our data for training and 25% for testing.

The next step is to instantiate `ShallowNet`, followed by training the network itself:

```
44 # initialize the optimizer and model
45 print("[INFO] compiling model...")
46 opt = SGD(lr=0.005)
47 model = ShallowNet.build(width=32, height=32, depth=3, classes=3)
48 model.compile(loss="categorical_crossentropy", optimizer=opt,
49     metrics=["accuracy"])
50
51 # train the network
52 print("[INFO] training network...")
53 H = model.fit(trainX, trainY, validation_data=(testX, testY),
54     batch_size=32, epochs=100, verbose=1)
```

We initialize the `SGD` optimizer on **Line 46** using a learning rate of `0.005` (we'll discuss how to tune learning rates in a future chapter). The `ShallowNet` architecture is instantiated on **Line 47**, supplying a width and height of 32 pixels along with a depth of 3 – this implies that our input images are 32×32 pixels with three channels. Since the Animals dataset has three class labels, we set `classes=3`.

The `model` is then compiled on **Lines 48 and 49** where we'll use cross-entropy as our loss function and SGD as our optimizer. To actual train the network, we make a call to the `.fit` method of `model` on **Lines 53 and 54**. The `.fit` method requires us to pass in the training and testing data. We'll also supply our testing data so we can evaluate the performance of ShallowNet after each epoch. The network will be trained for 100 epochs using mini-batch sizes of 32 (meaning that 32 images will be presented to the network at a time, and a full forward and backward pass will be done to update the parameters of the network).

After training our network, we can evaluate its performance:

```
56 # evaluate the network
57 print("[INFO] evaluating network...")
58 predictions = model.predict(testX, batch_size=32)
59 print(classification_report(testY.argmax(axis=1),
60     predictions.argmax(axis=1),
61     target_names=["cat", "dog", "panda"]))
```

To obtain the output predictions on our testing data, we call `.predict` of the `model`. A nicely formatted classification report is displayed to our screen on **Lines 59-61**.

Our final code block handles plotting the accuracy and loss over time for *both* the training and testing data:

```
63 # plot the training loss and accuracy
64 plt.style.use("ggplot")
65 plt.figure()
66 plt.plot(np.arange(0, 100), H.history["loss"], label="train_loss")
67 plt.plot(np.arange(0, 100), H.history["val_loss"], label="val_loss")
68 plt.plot(np.arange(0, 100), H.history["acc"], label="train_acc")
69 plt.plot(np.arange(0, 100), H.history["val_acc"], label="val_acc")
70 plt.title("Training Loss and Accuracy")
71 plt.xlabel("Epoch #")
72 plt.ylabel("Loss/Accuracy")
73 plt.legend()
74 plt.show()
```

To train ShallowNet on the Animals dataset, just execute the following command:

```
$ python shallownet_animals.py --dataset ../datasets/animals
```

Training should be quite fast as the network is *very* shallow and our image dataset is relatively small:

```
[INFO] loading images...
[INFO] processed 500/3000
[INFO] processed 1000/3000
[INFO] processed 1500/3000
[INFO] processed 2000/3000
[INFO] processed 2500/3000
[INFO] processed 3000/3000
[INFO] compiling model...
[INFO] training network...
Train on 2250 samples, validate on 750 samples
Epoch 1/100
0s - loss: 1.0290 - acc: 0.4560 - val_loss: 0.9602 - val_acc: 0.5160
Epoch 2/100
0s - loss: 0.9289 - acc: 0.5431 - val_loss: 1.0345 - val_acc: 0.4933
...
Epoch 100/100
0s - loss: 0.3442 - acc: 0.8707 - val_loss: 0.6890 - val_acc: 0.6947
[INFO] evaluating network...
             precision    recall  f1-score   support

        cat       0.58      0.77      0.67       239
        dog       0.75      0.40      0.52       249
      panda       0.79      0.90      0.84       262

avg / total       0.71      0.69      0.68       750
```

Due to the small amount of training data, epochs were quite speedy, taking less than one second on both my CPU and GPU.

As you can see from the output above, ShallowNet obtained 71% ***classification accuracy*** on our testing data, a massive improvement from our previous best of 59% using simple feedforward neural networks. Using more advanced training networks, as well as a more powerful architecture, we'll be able to boost classification accuracy even higher.

The loss and accuracy plotted over time is displayed in Figure 12.2. On the x-axis we have our epoch number and on the y-axis we have our loss and accuracy. Examining this figure, we can see that learning is a bit volatile with large spikes in loss around epoch 20 and epoch 60 – this result is likely due to our learning rate being too high, something we'll help resolve in Chapter 16.

Also take note that the training and testing loss diverge heavily past epoch 30, which implies that our network is modeling the training data *too closely* and overfitting. We can remedy this issue by obtaining more data or applying techniques like data augmentation (covered in the *Practitioner Bundle*).

Around epoch 60 our testing accuracy saturates – we are unable to get past $\approx 70\%$ classification accuracy, meanwhile our training accuracy continues to climb to over 85%. Again, gathering more training data, applying data augmentation, and taking more care to tune our learning rate will help us improve our results in the future.

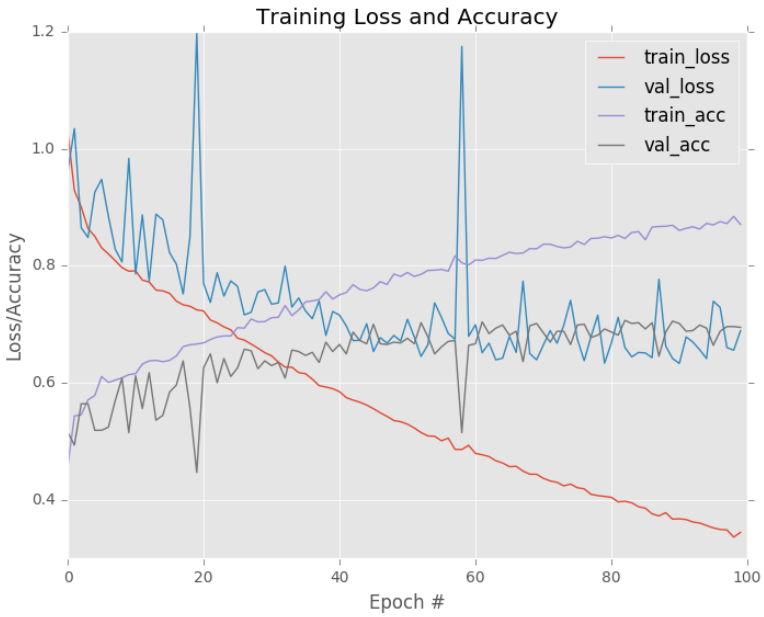

Figure 12.2: A plot of our loss and accuracy over the course of 100 epochs for the ShallowNet architecture trained on the Animals dataset.

The key point here is that an *extremely simple* Convolutional Neural Network was able to obtain 71% classification accuracy on the Animals dataset where our previous best was only 59% – that's an improvement of over 12%!

12.2.3 ShallowNet on CIFAR-10

Let's also apply the ShallowNet architecture to the CIFAR-10 dataset to see if we can improve our results. Open a new file, name it `shallownet_cifar10.py`, and insert the following code:

```
# import the necessary packages
from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import classification_report
from pyimagesearch.nn.conv import ShallowNet
from keras.optimizers import SGD
from keras.datasets import cifar10
import matplotlib.pyplot as plt
import numpy as np

# load the training and testing data, then scale it into the
# range [0, 1]
print("[INFO] loading CIFAR-10 data...")
((trainX, trainY), (testX, testY)) = cifar10.load_data()
trainX = trainX.astype("float") / 255.0
testX = testX.astype("float") / 255.0

# convert the labels from integers to vectors
lb = LabelBinarizer()
```

```
19 trainY = lb.fit_transform(trainY)
20 testY = lb.transform(testY)
21
22 # initialize the label names for the CIFAR-10 dataset
23 labelNames = ["airplane", "automobile", "bird", "cat", "deer",
24     "dog", "frog", "horse", "ship", "truck"]
```

Lines 2-8 import our required Python packages. We then load the CIFAR-10 dataset (pre-split into training and testing sets), followed by scaling the image pixel intensities to the range $[0, 1]$. Since the CIFAR-10 images are preprocessed and the channel ordering is handled *automatically* inside of `cifar10.load_data`, we do not need to apply any of our custom preprocessing classes.

Our labels are then one-hot encoded to vectors on **Lines 18-20**. We also initialize the label names for the CIFAR-10 dataset on **Lines 23 and 24**.

Now that our data is prepared, we can train ShallowNet:

```
26 # initialize the optimizer and model
27 print("[INFO] compiling model...")
28 opt = SGD(lr=0.01)
29 model = ShallowNet.build(width=32, height=32, depth=3, classes=10)
30 model.compile(loss="categorical_crossentropy", optimizer=opt,
31     metrics=["accuracy"])
32
33 # train the network
34 print("[INFO] training network...")
35 H = model.fit(trainX, trainY, validation_data=(testX, testY),
36     batch_size=32, epochs=40, verbose=1)
```

Line 28 initializes the SGD optimizer with a learning rate of `0.01`. ShallowNet is then constructed on **Line 29** using a width of 32, a height of 32, a depth of 3 (since CIFAR-10 images have three channels). We set `classes=10` since, as the name suggests, there are ten classes in the CIFAR-10 dataset. The model is compiled on **Lines 30 and 31** then trained on **Lines 35 and 36** over the course of 40 epochs.

Evaluating ShallowNet is done in the exact same manner as our previous example with the Animals dataset:

```
38 # evaluate the network
39 print("[INFO] evaluating network...")
40 predictions = model.predict(testX, batch_size=32)
41 print(classification_report(testY.argmax(axis=1),
42     predictions.argmax(axis=1), target_names=labelNames))
```

We'll also plot the loss and accuracy over time so we can get an idea how our network is performing:

```
44 # plot the training loss and accuracy
45 plt.style.use("ggplot")
46 plt.figure()
47 plt.plot(np.arange(0, 40), H.history["loss"], label="train_loss")
48 plt.plot(np.arange(0, 40), H.history["val_loss"], label="val_loss")
```

Figure 12.3: Loss and accuracy for ShallowNet trained on CIFAR-10. Our network obtains 60% classification accuracy; however, it is overfitting. Further accuracy can be obtained by applying regularization, which we'll cover later in this book.

```
49 plt.plot(np.arange(0, 40), H.history["acc"], label="train_acc")
50 plt.plot(np.arange(0, 40), H.history["val_acc"], label="val_acc")
51 plt.title("Training Loss and Accuracy")
52 plt.xlabel("Epoch #")
53 plt.ylabel("Loss/Accuracy")
54 plt.legend()
55 plt.show()
```

To train ShallowNet on CIFAR-10, simply execute the following command:

```
$ python shallownet_cifar10.py
[INFO] loading CIFAR-10 data...
[INFO] compiling model...
[INFO] training network...
Train on 50000 samples, validate on 10000 samples
Epoch 1/40
5s - loss: 1.8087 - acc: 0.3653 - val_loss: 1.6558 - val_acc: 0.4282
Epoch 2/40
5s - loss: 1.5669 - acc: 0.4583 - val_loss: 1.4903 - val_acc: 0.4724
...
Epoch 40/40
5s - loss: 0.6768 - acc: 0.7685 - val_loss: 1.2418 - val_acc: 0.5890
[INFO] evaluating network...
             precision    recall  f1-score   support
```

```
            airplane       0.62      0.68      0.65      1000
          automobile       0.79      0.64      0.71      1000
                bird       0.43      0.46      0.44      1000
                 cat       0.42      0.38      0.40      1000
                deer       0.52      0.51      0.52      1000
                 dog       0.44      0.57      0.50      1000
                frog       0.74      0.61      0.67      1000
               horse       0.71      0.61      0.66      1000
                ship       0.65      0.77      0.70      1000
               truck       0.67      0.66      0.66      1000

           avg / total       0.60      0.59      0.59     10000
```

Again, epochs are quite fast due to the shallow network architecture and relatively small dataset. Using my GPU, I obtained 5-second epochs while my CPU took 22 seconds for each epoch.

After 40 epochs ShallowNet is evaluated and we find that it obtains **60% accuracy** on the testing set, an increase from the previous 57% accuracy using simple neural networks.

More importantly, plotting our loss and accuracy in Figure 12.3 gives us some insight to the training process demonstrates that our validation loss does not skyrocket. Our training and testing loss/accuracy start to diverge past epoch 10. Again, this can be attributed to a larger learning rate and the fact we aren't using methods to help combat overfitting (regularization parameters, dropout, data augmentation, etc.).

It is also *notoriously easy* to overfit on the CIFAR-10 dataset due to the limited number of low-resolution training samples. As we become more comfortable building and training our own custom Convolutional Neural Networks, we'll discover methods to boost classification accuracy on CIFAR-10 while simultaneously reducing overfitting.

12.3 Summary

In this chapter, we implemented our first Convolutional Neural Network architecture, ShallowNet, and trained it on the Animals and CIFAR-10 dataset. ShallowNet obtained 71% classification accuracy on Animals, an increase of 12% from our previous best using simple feedforward neural networks.

When applied to CIFAR-10, ShallowNet reached 60% accuracy, a whopping increase of 45% accuracy over our previous best.

ShallowNet is an *extremely* simple CNN that uses only *one* `CONV` layer – further accuracy can be obtained by training deeper networks with multiple sets of `CONV => RELU => POOL` operations.

This case study, *"Breaking Captchas with CNNs"* is from the *Starter Bundle*...

21. Case Study: Breaking Captchas with a CNN

So far in this book we've worked with datasets that have been pre-compiled and labeled for us – *but what if we wanted to go about creating our own* ***custom dataset*** *and then training a CNN on it?* In this chapter, I'll present a *complete* deep learning case study that will give you an example of:

1. Downloading a set of images.
2. Labeling and annotating your images for training.
3. Training a CNN on your custom dataset.
4. Evaluating and testing the trained CNN.

The dataset of images we'll be downloading is a set of captcha images used to prevent bots from automatically registering or logging in to a given website (or worse, trying to brute force their way into someone's account).

Once we've downloaded a set of captcha images we'll need to manually label each of the digits in the captcha. As we'll find out, *obtaining* and *labeling* a dataset can be half (if not more) the battle. Depending on how much data you need, how easy it is to obtain, and whether or not you need to label the data (i.e., assign a ground-truth label to the image), it can be a costly process, both in terms of time and/or finances (if you pay someone else to label the data).

Therefore, whenever possible we try to use traditional computer vision techniques to speedup the labeling process. In the context of this chapter, if we were to use image processing software such as Photoshop or GIMP to manually extract digits in a captcha image to create our training set, it might takes us *days* of non-stop work to complete the task.

However, by applying some basic computer vision techniques, we can download and label our training set in *less than an hour*. This is one of the many reasons why I encourage deep learning practitioners to also invest in their computer vision education. Books such as *Practical Python and OpenCV* are meant to help you master the fundamentals of computer vision and OpenCV quickly – if you are serious about mastering deep learning applied to computer vision, you would do well to learn the basics of the broader computer vision and image processing field as well.

I'd also like to mention that datasets in the real-world are not like the benchmark datasets such as MNIST, CIFAR-10, and ImageNet where images are neatly labeled and organized and our goal is only to train a model on the data and evaluate it. These benchmark datasets may be

challenging, but in the real-world, *the struggle is often obtaining the (labeled) data itself* – and in many instances, the labeled data is worth *a lot more* than the deep learning model obtained from training a network on your dataset.

For example, if you were running a company responsible for creating a custom Automatic License Plate Recognition (ANPR) system for the United States government, you might invest *years* building a robust, massive dataset, while at the same time evaluating various deep learning approaches to recognizing license plates. Accumulating such a massive labeled dataset would give you a competitive edge over other companies – and in this case, the *data itself* is worth more than the end product.

Your company would be more likely to be acquired simply because of the *exclusive* rights you have to the massive, labeled dataset. Building an amazing deep learning model to recognize license plates would only increase the value of your company, but again, *labeled data* is expensive to obtain and replicate, so if you own the keys to a dataset that is hard (if not impossible) to replicate, make no mistake: your company's primary asset is the data, not the deep learning.

In the remainder of this chapter, we'll look how we can obtain a dataset of images, label them, and then apply deep learning to break a captcha system.

21.1 Breaking Captchas with a CNN

This chapter is broken into many parts to help keep it organized and easy to read. In the first section I discuss the captcha dataset we are working with and discuss the concept of **responsible disclosure** – something you should *always* do when computer security is involved.

From there I discuss the directory structure of our project. We then create a Python script to *automatically* download a set of images that we'll be using for training and evaluation.

After downloading our images, we'll need to use a bit of computer vision to aid us in labeling the images, making the process *much easier* and *substantially faster* than simply cropping and labeling inside photo software like GIMP or Photoshop. Once we have labeled our data, we'll train the LeNet architecture – as we'll find out, we're able to break the captcha system and obtain 100% accuracy in less than 15 epochs.

21.1.1 A Note on Responsible Disclosure

Living in the northeastern/midwestern part of the United States, it's hard to travel on major highways without an E-ZPass [133]. E-ZPass is an electronic toll collection system used on many bridges, interstates, and tunnels. Travelers simply purchase an E-ZPass transponder, place it on the windshield of their car, and enjoy the ability to quickly travel through tolls without stopping, as a credit card attached to their E-ZPass account is charged for any tolls.

E-ZPass has made tolls a much more "enjoyable" process (if there is such a thing). Instead of waiting in interminable lines where a physical transaction needs to take place (i.e., hand the cashier money, receive your change, get a printed receipt for reimbursement, etc.), you can simply blaze through in the fast lane without stopping – it saves a bunch of time when traveling and is much less of a hassle (you still have to pay the toll though).

I spend much of my time traveling between Maryland and Connecticut, two states along the I-95 corridor of the United States. The I-95 corridor, especially in New Jersey, contains a plethora of toll booths, so an E-ZPass pass was a no-brainer decision for me. About a year ago, the credit card I had attached to my E-ZPass account expired, and I needed to update it. I went to the E-ZPass New York website (the state I bought my E-ZPass in) to log in and update my credit card, but I stopped dead in my tracks (Figure 21.1).

Can you spot the flaw in this system? Their "captcha" is nothing more than four digits on a plain white background which is a major security risk – someone with even basic computer vision

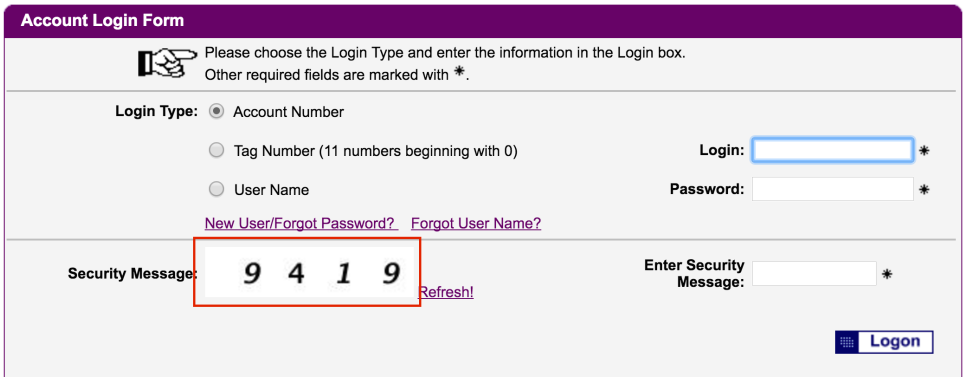

Figure 21.1: The E-Z Pass New York login form. Can you spot the flaw in their login system?

or deep learning experience could develop a piece of software to break this system.

This is where the concept of ***responsible disclosure*** comes in. Responsible disclosure is a computer security term for describing how to disclose a vulnerability. Instead of posting it on the internet for everyone to see *immediately* after the threat is detected, you try to contact the stakeholders first to ensure they know there is an issue. The stakeholders can then attempt to patch the software and resolve the vulnerability.

Simply ignoring the vulnerability and hiding the issue is a *false security*, something that should be avoided. In an ideal world, the vulnerability is resolved *before* it is publicly disclosed.

However, when stakeholders do not acknowledge the issue or do not fix the problem in a reasonable amount of time it creates an ethical conundrum – do you hide the issue and pretend it doesn't exist? Or do you disclose it, bringing more attention to the problem in an effort to bring a fix to the problem faster? Responsible disclosure states that you first bring the problem to the stakeholders (*responsible*) – if it's not resolved, then you need to disclose the issue (*disclosure*).

To demonstrate how the E-ZPass NY system was at risk, I trained a deep learning model to recognize the digits in the captcha. I then wrote a second Python script to (1) auto-fill my login credentials and (2) break the captcha, allowing my script access to my account.

In this case, I was only auto-logging into my account. Using this "feature", I could auto-update a credit card, generate reports on my tolls, or even add a new car to my E-ZPass. But someone nefarious may use this as a method to brute force their way into a customer's account.

I contacted E-ZPass over email, phone, and Twitter regarding the issue ***one year before*** I wrote this chapter. They acknowledged the receipt of my messages; however, nothing has been done to fix the issue, despite multiple contacts.

In the rest of this chapter, I'll discuss how we can use the E-ZPass system to obtain a captcha dataset which we'll then label and train a deep learning model on. I will *not* be sharing the Python code to auto-login to an account – that is outside the boundaries of responsible disclosure so please do not ask me for this code.

My honest hope is by the time this book is published that E-ZPass NY will have updated their website and resolved the captcha vulnerability, thereby leaving this chapter as a great example of applying deep learning to a hand-labeled dataset, with zero vulnerability threat.

Keep in mind that with all knowledge comes responsibility. This knowledge, *under no circumstance*, should be used for nefarious or unethical reasons. This case study exists as a method to demonstrate how to obtain and label a custom dataset, followed by training a deep learning model on top of it.

I am required to say that I am *not responsible* for how this code is used – use this as

an opportunity to learn, not an opportunity to be nefarious.

21.1.2 The Captcha Breaker Directory Structure

In order to build the captcha breaker system, we'll need to update the `pyimagesearch.utils` sub-module and include a new file named `captchahelper.py`:

```
|--- pyimagesearch
|    |--- __init__.py
|    |--- datasets
|    |--- nn
|    |--- preprocessing
|    |--- utils
|    |    |--- __init__.py
|    |    |--- captchahelper.py
```

This file will store a utility function named `preprocess` to help us process digits before feeding them into our deep neural network.

We'll also create a second directory, this one named `captcha_breaker`, outside of our *pyimagesearch* module, and include the following files and subdirectories:

```
|--- captcha_breaker
|    |--- dataset/
|    |--- downloads/
|    |--- output/
|    |--- annotate.py
|    |--- download_images.py
|    |--- test_model.py
|    |--- train_model.py
```

The `captcha_breaker` directory is where all our project code will be stored to break image captchas. The `dataset` directory is where we will store our *labeled* digits which we'll be hand-labeling. I prefer to keep my datasets organized using the following directory structure template:

```
root_directory/class_name/image_filename.jpg
```

Therefore, our `dataset` directory will have the structure:

```
dataset/{1-9}/example.jpg
```

Where `dataset` is the root directory, `{1-9}` are the possible digit names, and `example.jpg` will be an example of the given digit.

The `downloads` directory will store the raw captcha .jpg files downloaded from the E-ZPass website. Inside the `output` directory, we'll store our trained LeNet architecture.

The `download_images.py` script, as the name suggests, will be responsible for actually downloading the example captchas and saving them to disk. Once we've downloaded a set of captchas we'll need to extract the digits from each image and hand-label every digit – this will be accomplished by `annotate.py`.

The `train_model.py` script will train LeNet on the labeled digits while `test_model.py` will apply LeNet to captcha images themselves.

21.1.3 Automatically Downloading Example Images

The first step in building our captcha breaker is to download the example captcha images themselves. If we were to right click on the captcha image next to the text *"Security Image"* in Figure 21.1 above, we would obtain the following URL:

`https://www.e-zpassny.com/vector/jcaptcha.do`

If you copy and paste this URL into your web browser and hit refresh multiple times, you'll notice that this is a dynamic program that generates a new captcha each time you refresh. Therefore, to obtain our example captcha images we need to request this image a few hundred times and save the resulting image.

To automatically fetch new captcha images and save them to disk we can use `download_images.py`:

```
1  # import the necessary packages
2  import argparse
3  import requests
4  import time
5  import os
6
7  # construct the argument parse and parse the arguments
8  ap = argparse.ArgumentParser()
9  ap.add_argument("-o", "--output", required=True,
10     help="path to output directory of images")
11 ap.add_argument("-n", "--num-images", type=int,
12     default=500, help="# of images to download")
13 args = vars(ap.parse_args())
```

Lines 2-5 import our required Python packages. The `requests` library makes working with HTTP connections easy and is heavily used in the Python ecosystem. If you do not already have `requests` installed on your system, you can install it via:

```
$ pip install requests
```

We then parse our command line arguments on **Lines 8-13**. We'll require a single command line argument, `--output`, which is the path to the output directory that will store our raw captcha images (we'll later hand label each of the digits in the images).

A second optional switch `--num-images`, controls the number of captcha images we're going to download. We'll default this value to 500 total images. Since there are four digits in each captcha, this value of 500 will give us $500 \times 4 = 2,000$ total digits that we can use for training our network.

Our next code block initializes the URL of the captcha image we are going to download along with the total number of images generated thus far:

```
15 # initialize the URL that contains the captcha images that we will
16 # be downloading along with the total number of images downloaded
17 # thus far
18 url = "https://www.e-zpassny.com/vector/jcaptcha.do"
19 total = 0
```

We are now ready to download the captcha images:

```
21 # loop over the number of images to download
22 for i in range(0, args["num_images"]):
```

```
23      try:
24          # try to grab a new captcha image
25          r = requests.get(url, timeout=60)
26
27          # save the image to disk
28          p = os.path.sep.join([args["output"], "{}.jpg".format(
29              str(total).zfill(5))])
30          f = open(p, "wb")
31          f.write(r.content)
32          f.close()
33
34          # update the counter
35          print("[INFO] downloaded: {}".format(p))
36          total += 1
37
38      # handle if any exceptions are thrown during the download process
39      except:
40          print("[INFO] error downloading image...")
41
42      # insert a small sleep to be courteous to the server
43      time.sleep(0.1)
```

On **Line 22** we start looping over the `--num-images` that we wish to download. A request is made on **Line 25** to download the image. We then save the image to disk on **Lines 28-32**. If there was an error downloading the image, our `try/except` block on **Line 39 and 40** catches it and allows our script to continue. Finally, we insert a small sleep on **Line 43** to be courteous to the web server we are requesting.

You can execute `download_images.py` using the following command:

```
$ python download_images.py --output downloads
```

This script will take awhile to run since we have (1) are making a network request to download the image and (2) inserted a 0.1 second pause after each download.

Once the program finishes executing you'll see that your `download` directory is filled with images:

```
$ ls -l downloads/*.jpg | wc -l
500
```

However, there are just the *raw captcha images* – we need to *extract* and *label* each of the digits in the captchas to create our training set. To accomplish this, we'll use a bit of OpenCV and image processing techniques to make our life easier.

21.1.4 Annotating and Creating Our Dataset

So, how do you go about labeling and annotating each of our captcha images? Do we open up Photoshop or GIMP and use the "select/marquee" tool to copy out a given digit, save it to disk, and then repeat *ad nauseam*? If we did, it might take us *days* of non-stop working to label each of the digits in the raw captcha images.

Instead, a better approach would be to use basic image processing techniques inside the OpenCV library to help us out. To see how we can label our dataset more efficiently, open a new file, name it `annotate.py`, and inserting the following code:

```
1  # import the necessary packages
2  from imutils import paths
3  import argparse
4  import imutils
5  import cv2
6  import os
7
8  # construct the argument parse and parse the arguments
9  ap = argparse.ArgumentParser()
10 ap.add_argument("-i", "--input", required=True,
11      help="path to input directory of images")
12 ap.add_argument("-a", "--annot", required=True,
13      help="path to output directory of annotations")
14 args = vars(ap.parse_args())
```

Lines 2-6 import our required Python packages while **Lines 9-14** parse our command line arguments. This script requires two arguments:

- `--input`: The input path to our raw captcha images (i.e., the `downloads` directory).
- `--annot`: The output path to where we'll be storing the labeled digits (i.e., the `dataset` directory).

Our next code block grabs the paths to all images in the `--input` directory and initializes a dictionary named `counts` that will store the total number of times a given digit (the key) has been labeled (the value):

```
16 # grab the image paths then initialize the dictionary of character
17 # counts
18 imagePaths = list(paths.list_images(args["input"]))
19 counts = {}
```

The actual annotation process starts below:

```
21 # loop over the image paths
22 for (i, imagePath) in enumerate(imagePaths):
23      # display an update to the user
24      print("[INFO] processing image {}/{}".format(i + 1,
25           len(imagePaths)))
26
27      try:
28           # load the image and convert it to grayscale, then pad the
29           # image to ensure digits caught on the border of the image
30           # are retained
31           image = cv2.imread(imagePath)
32           gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
33           gray = cv2.copyMakeBorder(gray, 8, 8, 8, 8,
34                cv2.BORDER_REPLICATE)
```

On **Line 22** we start looping over each of the individual `imagePaths`. For each image, we load it from disk (**Line 31**), convert it to grayscale (**Line 32**), and pad the borders of the image with eight pixels in every direction (**Line 33 and 34**). Figure 21.2 below shows the difference between the original image (*left*) and the padded image (*right*).

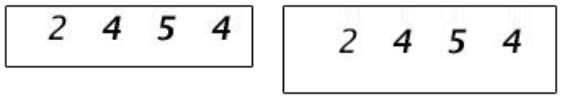

Figure 21.2: **Left:** The original image loaded from disk. **Right:** Padding the image to ensure we can extract the digits *just in case* any of the digits are touching the border of the image.

We perform this padding *just in case* any of our digits are touching the border of the image. If the digits *were* touching the border, we wouldn't be able to extract them from the image. Thus, to prevent this situation, we purposely pad the input image so it's *not possible* for a given digit to touch the border.

We are now ready to binarize the input image via Otsu's thresholding method (Chapter 9, *Practical Python and OpenCV*):

```
36             # threshold the image to reveal the digits
37             thresh = cv2.threshold(gray, 0, 255,
38                 cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU)[1]
```

This function call automatically thresholds our image such that our image is now *binary* – black pixels represent the *background* while white pixels are our *foreground* as shown in Figure 21.3.

Figure 21.3: Thresholding the image ensures the foreground is *white* while the background is *black*. This is a typical assumption/requirement when working with many image processing functions with OpenCV.

Thresholding the image is a critical step in our image processing pipeline as we now need to find the *outlines* of each of the digits:

```
40             # find contours in the image, keeping only the four largest
41             # ones
42             cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
43                 cv2.CHAIN_APPROX_SIMPLE)
44             cnts = cnts[0] if imutils.is_cv2() else cnts[1]
45             cnts = sorted(cnts, key=cv2.contourArea, reverse=True)[:4]
```

Lines 42 and 43 find the contours (i.e., outlines) of each of the digits in the image. Just in case there is "noise" in the image we sort the contours by their area, keeping only the four largest one (i.e., our digits themselves).

Given our contours we can extract each of them by computing the bounding box:

```
47             # loop over the contours
48             for c in cnts:
```

```
49              # compute the bounding box for the contour then extract
50              # the digit
51              (x, y, w, h) = cv2.boundingRect(c)
52              roi = gray[y - 5:y + h + 5, x - 5:x + w + 5]
53
54              # display the character, making it larger enough for us
55              # to see, then wait for a keypress
56              cv2.imshow("ROI", imutils.resize(roi, width=28))
57              key = cv2.waitKey(0)
```

On **Line 48** we loop over each of the contours found in the thresholded image. We call `cv2.boundingRect` to compute the bounding box (x,y)-coordinates of the digit region. This region of interest (ROI) is then extracted from the grayscale image on **Line 52**. I have included a sample of example digits extracted from their raw captcha images as a montage in Figure 21.4.

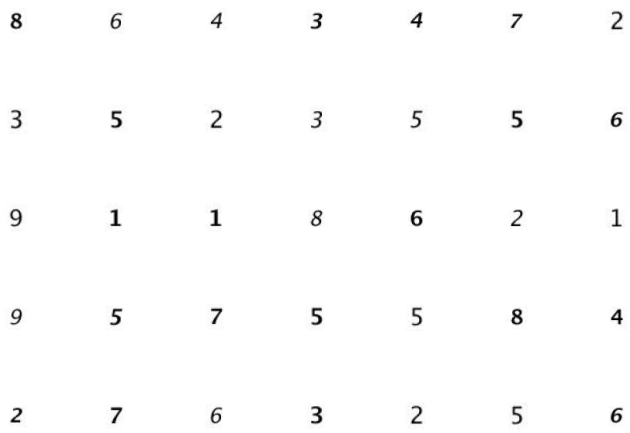

Figure 21.4: A sample of the digit ROIs extracted from our captcha images. Our goal will be to label these images in such a way that we can train a custom Convolutional Neural Network on them.

Line 56 displays the digit ROI to our screen, resizing it to be large enough for us to see easily. **Line 57** then waits for a keypress on your keyboard – but choose your keypress wisely! The key you press will be used as the *label* for the digit.

To see how the labeling process works via the `cv2.waitKey` call, take a look at the following code block:

```
59              # if the '`' key is pressed, then ignore the character
60              if key == ord("`"):
61                  print("[INFO] ignoring character")
62                  continue
63
64              # grab the key that was pressed and construct the path
65              # the output directory
66              key = chr(key).upper()
67              dirPath = os.path.sep.join([args["annot"], key])
68
```

```
69                # if the output directory does not exist, create it
70                if not os.path.exists(dirPath):
71                     os.makedirs(dirPath)
```

If the tilde key ‘ (tilde) is pressed, we'll ignore the character (**Lines 60 and 62**). Needing to ignore a character may happen if our script accidentally detects "noise" (i.e., anything but a digit) in the input image or if we are not sure what the digit is. Otherwise, we assume that the key pressed was the *label* for the digit (**Line 66**) and use the key to construct the directory path to our output label (**Line 67**).

For example, if I pressed the 7 key on my keyboard, the `dirPath` would be:

```
dataset/7
```

Therefore, all images containing the digit "7" will be stored in the `dataset/7` sub-directory. **Lines 70 and 71** make a check to see if the `dirPath` directory does not exist – if it doesn't, we create it.

Once we have ensured that `dirPath` properly exists, we simply have to write the example digit to file:

```
73                  # write the labeled character to file
74                count = counts.get(key, 1)
75                p = os.path.sep.join([dirPath, "{}.png".format(
76                     str(count).zfill(6))])
77                cv2.imwrite(p, roi)
78
79                # increment the count for the current key
80                counts[key] = count + 1
```

Line 74 grabs the total number of examples written to disk thus far for the current digit. We then construct the output path to the example digit using the `dirPath`. After executing **Lines 75 and 76**, our output path `p` may look like:

```
datasets/7/000001.png
```

Again, notice how all example ROIs that contain the number seven will be stored in the `datasets/7` subdirectory – this is an easy, convenient way to organize your datasets when labeling images.

Our final code block handles if we want to `ctrl+c` out of the script to exit *or* if there is an error processing an image:

```
82          # we are trying to control-c out of the script, so break from the
83          # loop (you still need to press a key for the active window to
84          # trigger this)
85          except KeyboardInterrupt:
86               print("[INFO] manually leaving script")
87               break
88
89          # an unknown error has occurred for this particular image
90          except:
91               print("[INFO] skipping image...")
```

If we wish to `ctrl+c` and quit the script early, **Line 85** detects this and allows our Python program to exit gracefully. **Line 90** catches *all other errors* and simply ignores them, allowing us to continue with the labeling process.

The *last* thing you want when labeling a dataset is for a random error to occur due to an image encoding problem, causing your entire program to crash. If this happens, you'll have to restart the labeling process all over again. You can obviously build in extra logic to detect where you left off, but such an example is outside the scope of this book.

To label the images you downloaded from the E-ZPass NY website, just execute the following command:

```
$ python annotate.py --input downloads --annot dataset
```

Here you can see that the number 7 is displayed to my screen in Figure 21.5.

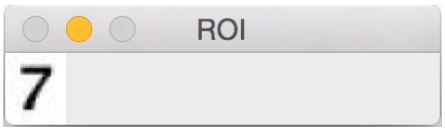

Figure 21.5: When annotating our dataset of digits, a given digit ROI will display on our screen. We then need to press the corresponding key on our keyboard to label the image and save the ROI to disk.

I then press 7 key on my keyboard to label it and then the digit is written to file in the `dataset/7` sub-directory.

The `annotate.py` script then proceeds to the next digit for me to label. You can then proceed to label all of the digits in the raw captcha images. You'll quickly realize that labeling a dataset can be very tedious, time-consuming process. Labeling all 2,000 digits should take you less than half an hour – but you'll likely become bored within the first five minutes.

Remember, actually *obtaining* your labeled dataset is half the battle. From there the actual work can start. Luckily, I have already labeled the digits for you! If you check the `dataset` directory included in the accompanying downloads of this book you'll find the entire dataset ready to go:

```
$ ls dataset/
1  2  3  4  5  6  7  8  9
$ ls -l dataset/1/*.png | wc -l
232
```

Here you can see nine sub-directories, one for each of the digits that we wish to recognize. Inside each subdirectory, there are example images of the particular digit. Now that we have our labeled dataset, we can proceed to training our captcha breaker using the LeNet architecture.

21.1.5 Preprocessing the Digits

As we know, our Convolutional Neural Networks require an image with a fixed width and height to be passed in during training. However, our labeled digit images are of various sizes – some are taller than they are wide, others are wider than they are tall. Therefore, we need a method to pad and resize our input images to a fixed size *without* distorting their aspect ratio.

We can resize and pad our images while preserving the aspect ratio by defining a `preprocess` function inside `captchahelper.py`:

```
1  # import the necessary packages
2  import imutils
3  import cv2
4
5  def preprocess(image, width, height):
6       # grab the dimensions of the image, then initialize
7       # the padding values
8       (h, w) = image.shape[:2]
9
10      # if the width is greater than the height then resize along
11      # the width
12      if w > h:
13           image = imutils.resize(image, width=width)
14
15      # otherwise, the height is greater than the width so resize
16      # along the height
17      else:
18           image = imutils.resize(image, height=height)
```

Our `preprocess` function requires three parameters:

1. `image`: The input image that we are going to pad and resize.
2. `width`: The target output width of the image.
3. `height`: The target output height of the image.

On **Lines 12 and 13** we make a check to see if the width and is greater than the height, and if so, we resize the image along the larger dimension (width) Otherwise, if the height is greater than the width, we resize along the height (**Lines 17 and 18**), which implies either the width or height (depending on the dimensions of the input image) are fixed.

However, the opposite dimension is smaller than it should be. To fix this issue, we can "pad" the image along the shorter dimension to obtain our fixed size:

```
20        # determine the padding values for the width and height to
21      # obtain the target dimensions
22      padW = int((width - image.shape[1]) / 2.0)
23      padH = int((height - image.shape[0]) / 2.0)
24
25      # pad the image then apply one more resizing to handle any
26      # rounding issues
27      image = cv2.copyMakeBorder(image, padH, padH, padW, padW,
28           cv2.BORDER_REPLICATE)
29      image = cv2.resize(image, (width, height))
30
31      # return the pre-processed image
32      return image
```

Lines 22 and 23 compute the required amount of padding to reach the target `width` and `height`. **Lines 27 and 28** apply the padding to the image. Applying this padding should bring our image to our target `width` and `height`; however, there may be cases where we are one pixel off in a given dimension. The easiest way to resolve this discrepancy is to simply call `cv2.resize` (**Line 29**) to ensure all images are the same width and height.

The reason we do not *immediately* call `cv2.resize` at the top of the function is because we first need to consider the aspect ratio of the input image and attempt to pad it correctly first. If we do not maintain the image aspect ratio, then our digits will become distorted.

21.1.6 Training the Captcha Breaker

Now that our `preprocess` function is defined, we can move on to training LeNet on the image captcha dataset. Open up the `train_model.py` file and insert the following code:

```
1  # import the necessary packages
2  from sklearn.preprocessing import LabelBinarizer
3  from sklearn.model_selection import train_test_split
4  from sklearn.metrics import classification_report
5  from keras.preprocessing.image import img_to_array
6  from keras.optimizers import SGD
7  from pyimagesearch.nn.conv import LeNet
8  from pyimagesearch.utils.captchahelper import preprocess
9  from imutils import paths
10 import matplotlib.pyplot as plt
11 import numpy as np
12 import argparse
13 import cv2
14 import os
```

Lines 2-14 import our required Python packages. Notice that we'll be using the SGD optimizer along with the LeNet architecture to train a model on the digits. We'll also be using our newly defined `preprocess` function on each digit before passing it through our network.

Next, let's review our command line arguments:

```
16 # construct the argument parse and parse the arguments
17 ap = argparse.ArgumentParser()
18 ap.add_argument("-d", "--dataset", required=True,
19      help="path to input dataset")
20 ap.add_argument("-m", "--model", required=True,
21      help="path to output model")
22 args = vars(ap.parse_args())
```

The `train_model.py` script requires two command line arguments:

1. `--dataset`: The path to the input dataset of labeled captcha digits (i.e., the `dataset` directory on disk).
2. `--model`: Here we supply the path to where our serialized LeNet weights will be saved after training.

We can now load our data and corresponding labels from disk:

```
24 # initialize the data and labels
25 data = []
26 labels = []
27
28 # loop over the input images
29 for imagePath in paths.list_images(args["dataset"]):
30      # load the image, pre-process it, and store it in the data list
31      image = cv2.imread(imagePath)
32      image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
33      image = preprocess(image, 28, 28)
34      image = img_to_array(image)
35      data.append(image)
```

```
36
37      # extract the class label from the image path and update the
38      # labels list
39      label = imagePath.split(os.path.sep)[-2]
40      labels.append(label)
```

On **Lines 25 and 26** we initialize our `data` and `labels` lists, respectively. We then loop over every image in our labeled `--dataset` on **Line 29**. For each image in the dataset, we load it from disk, convert it to grayscale, and preprocess it such that it has a width of 28 pixels and a height of 28 pixels (**Lines 31-35**). The image is then converted to a Keras-compatible array and added to the `data` list (**Lines 34 and 35**).

One of the primary benefits of organizing your dataset directory structure in the format of:

```
root_directory/class_label/image_filename.jpg
```

is that you can easily extract the class label by grabbing the second-to-last component from the filename (**Line 39**). For example, given the input path `dataset/7/000001.png`, the `label` would be 7, which is the added to the `labels` list (**Line 40**).

Our next code block handles normalizing raw pixel intensity values to the range $[0, 1]$, followed by constructing the training and testing splits, along with one-hot encoding the labels:

```
42  # scale the raw pixel intensities to the range [0, 1]
43  data = np.array(data, dtype="float") / 255.0
44  labels = np.array(labels)
45
46  # partition the data into training and testing splits using 75% of
47  # the data for training and the remaining 25% for testing
48  (trainX, testX, trainY, testY) = train_test_split(data,
49      labels, test_size=0.25, random_state=42)
50
51  # convert the labels from integers to vectors
52  lb = LabelBinarizer().fit(trainY)
53  trainY = lb.transform(trainY)
54  testY = lb.transform(testY)
```

We can then initialize the LeNet model and SGD optimizer:

```
56  # initialize the model
57  print("[INFO] compiling model...")
58  model = LeNet.build(width=28, height=28, depth=1, classes=9)
59  opt = SGD(lr=0.01)
60  model.compile(loss="categorical_crossentropy", optimizer=opt,
61      metrics=["accuracy"])
```

Our input images will have a width of 28 pixels, a height of 28 pixels, and a single channel. There are a total of 9 digit classes we are recognizing (there is no 0 class).

Given the initialized model and optimizer we can train the network for 15 epochs, evaluate it, and serialize it to disk:

```
# train the network
print("[INFO] training network...")
H = model.fit(trainX, trainY,  validation_data=(testX, testY),
     batch_size=32, epochs=15, verbose=1)

# evaluate the network
print("[INFO] evaluating network...")
predictions = model.predict(testX, batch_size=32)
print(classification_report(testY.argmax(axis=1),
     predictions.argmax(axis=1), target_names=lb.classes_))

# save the model to disk
print("[INFO] serializing network...")
model.save(args["model"])
```

Our last code block will handle plotting the accuracy and loss for both the training and testing sets over time:

```
# plot the training + testing loss and accuracy
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, 15), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, 15), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, 15), H.history["acc"], label="acc")
plt.plot(np.arange(0, 15), H.history["val_acc"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
plt.show()
```

To train the LeNet architecture using the SGD optimizer on our custom captcha dataset, just execute the following command:

```
$ python train_model.py --dataset dataset --model output/lenet.hdf5
[INFO] compiling model...
[INFO] training network...
Train on 1509 samples, validate on 503 samples
Epoch 1/15
0s - loss: 2.1606 - acc: 0.1895 - val_loss: 2.1553 - val_acc: 0.2266
Epoch 2/15
0s - loss: 2.0877 - acc: 0.3565 - val_loss: 2.0874 - val_acc: 0.1769
Epoch 3/15
0s - loss: 1.9540 - acc: 0.5003 - val_loss: 1.8878 - val_acc: 0.3917
...
Epoch 15/15
0s - loss: 0.0152 - acc: 0.9993 - val_loss: 0.0261 - val_acc: 0.9980
[INFO] evaluating network...
             precision    recall  f1-score   support

          1       1.00      1.00      1.00        45
          2       1.00      1.00      1.00        55
```

```
              3       1.00      1.00      1.00        63
              4       1.00      0.98      0.99        52
              5       0.98      1.00      0.99        51
              6       1.00      1.00      1.00        70
              7       1.00      1.00      1.00        50
              8       1.00      1.00      1.00        54
              9       1.00      1.00      1.00        63

avg / total       1.00      1.00      1.00       503

[INFO] serializing network...
```

As we can see, after only 15 epochs our network is obtaining 100% classification accuracy on both the training and validation sets. This is not a case of overfitting either – when we investigate the training and validation curves in Figure 21.6 we can see that by epoch 5 the validation and training loss/accuracy match each other.

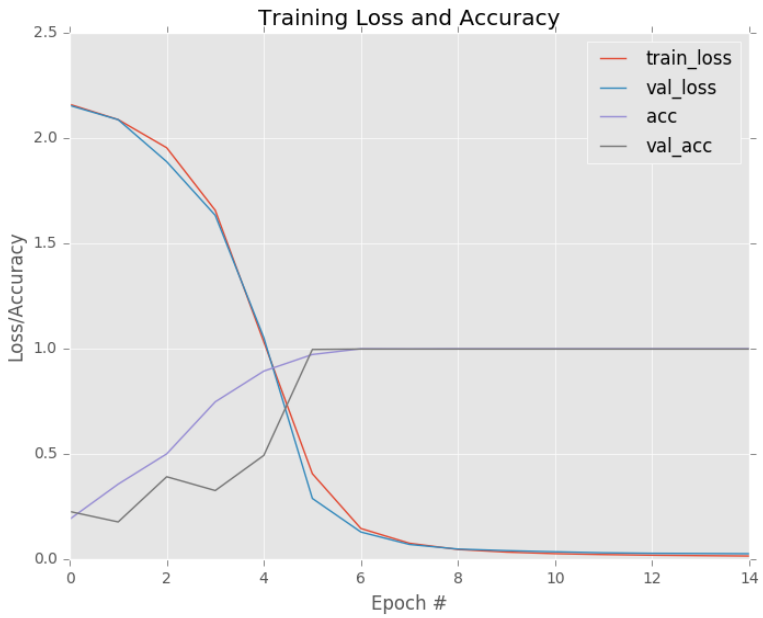

Figure 21.6: Using the LeNet architecture on our custom digits datasets enables us to obtain 100% classification accuracy after only fifteen epochs. Furthermore, there are no signs of overfitting.

If you check the `output` directory, you'll also see the serialized `lenet.hdf5` file:

```
$ ls -l output/
total 9844
-rw-rw-r-- 1 adrian adrian 10076992 May  3 12:56 lenet.hdf5
```

We can then use this model on new input images.

21.1.7 Testing the Captcha Breaker

Now that our captcha breaker is trained, let's test it out on some example images. Open up the `test_model.py` file and insert the following code:

```
# import the necessary packages
from keras.preprocessing.image import img_to_array
from keras.models import load_model
from pyimagesearch.utils.captchahelper import preprocess
from imutils import contours
from imutils import paths
import numpy as np
import argparse
import imutils
import cv2
```

As usual, our Python script starts with importing our Python packages. We'll again be using the `preprocess` function to prepare digits for classification.

Next, we'll parse our command line arguments:

```
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-i", "--input", required=True,
	help="path to input directory of images")
ap.add_argument("-m", "--model", required=True,
	help="path to input model")
args = vars(ap.parse_args())
```

The `--input` switch controls the path to the input captcha images that we wish to break. We could download a new set of captchas from the E-ZPass NY website, but for simplicity, we'll sample images from our existing raw captcha files. The `--model` argument is simply the path to the serialized weights residing on disk.

We can now load our pre-trained CNN and randomly sample ten captcha images to classify:

```
# load the pre-trained network
print("[INFO] loading pre-trained network...")
model = load_model(args["model"])

# randomly sample a few of the input images
imagePaths = list(paths.list_images(args["input"]))
imagePaths = np.random.choice(imagePaths, size=(10,),
	replace=False)
```

Here comes the fun part – actually breaking the captcha:

```
# loop over the image paths
for imagePath in imagePaths:
	# load the image and convert it to grayscale, then pad the image
	# to ensure digits caught only the border of the image are
	# retained
	image = cv2.imread(imagePath)
```

```
35      gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
36      gray = cv2.copyMakeBorder(gray, 20, 20, 20, 20,
37          cv2.BORDER_REPLICATE)
38
39      # threshold the image to reveal the digits
40      thresh = cv2.threshold(gray, 0, 255,
41          cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU)[1]
```

On **Line 30** we start looping over each of our sampled `imagePaths`. Just like in the `annotate.py` example, we need to extract each of the digits in the captcha. This extraction is accomplished by loading the image from disk, converting it to grayscale, and padding the border such that a digit cannot touch the boundary of the image (**Lines 34-37**). We add *extra padding* here so we have enough room to actually *draw* and *visualize* the correct prediction on the image.

Lines 40 and 41 threshold the image such that the digits appear as a *white foreground* against a *black background*.

We now need to find the contours of the digits in the `thresh` image:

```
43      # find contours in the image, keeping only the four largest ones,
44      # then sort them from left-to-right
45      cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
46          cv2.CHAIN_APPROX_SIMPLE)
47      cnts = cnts[0] if imutils.is_cv2() else cnts[1]
48      cnts = sorted(cnts, key=cv2.contourArea, reverse=True)[:4]
49      cnts = contours.sort_contours(cnts)[0]
50
51      # initialize the output image as a "grayscale" image with 3
52      # channels along with the output predictions
53      output = cv2.merge([gray] * 3)
54      predictions = []
```

We can find the digits by calling `cv2.findContours` on the `thresh` image. This function returns a list of (x,y)-coordinates that specify the *outline* of each individual digit.

We then perform two stages of sorting. The first stage sorts the contours by their *size*, keeping only the largest four outlines. We (correctly) assume that the four contours with the largest size are the digits we want to recognize. However, there is no guaranteed *spatial ordering* imposed on these contours – the third digit we wish to recognize may be first in the `cnts` list. Since we read digits from left-to-right, we need to sort the contours from left-to-right. This is accomplished via the `sort_contours` function (http://pyimg.co/sbm9p).

Line 53 takes our `gray` image and converts it to a three channel image by replicating the grayscale channel three times (one for each Red, Green, and Blue channel). We then initialize our list of `predictions` by the CNN on **Line 54**.

Given the contours of the digits in the captcha, we can now break it:

```
56      # loop over the contours
57      for c in cnts:
58          # compute the bounding box for the contour then extract the
59          # digit
60          (x, y, w, h) = cv2.boundingRect(c)
61          roi = gray[y - 5:y + h + 5, x - 5:x + w + 5]
62
63          # pre-process the ROI and classify it then classify it
```

```
64            roi = preprocess(roi, 28, 28)
65            roi = np.expand_dims(img_to_array(roi), axis=0)
66            pred = model.predict(roi).argmax(axis=1)[0] + 1
67            predictions.append(str(pred))
68
69            # draw the prediction on the output image
70            cv2.rectangle(output, (x - 2, y - 2),
71                 (x + w + 4, y + h + 4), (0, 255, 0), 1)
72            cv2.putText(output, str(pred), (x - 5, y - 5),
73                 cv2.FONT_HERSHEY_SIMPLEX, 0.55, (0, 255, 0), 2)
```

On **Line 57** we loop over each of the outlines (which have been sorted from left-to-right) of the digits. We then extract the ROI of the digit on **Lines 60 and 61** followed by preprocessing it on **Lines 64 and 65**.

Line 66 calls the `.predict` method of our `model`. The index with the *largest* probability returned by `.predict` will be our class label. We add `1` to this value since indexes values start at zero; however, there is no zero class – only classes for the digits 1-9. This prediction is then appended to the `predictions` list on **Line 67**.

Lines 70 and 71 draw a bounding box surrounding the current digit while **Lines 72 and 73** draw the predicted digit on the `output` image itself.

Our last code block handles writing the broken captcha as a string to our terminal as well as displaying the `output` image:

```
75        # show the output image
76      print("[INFO] captcha: {}".format("".join(predictions)))
77      cv2.imshow("Output", output)
78      cv2.waitKey()
```

To see our captcha breaker in action, simply execute the following command:

```
$ python test_model.py --input downloads --model output/lenet.hdf5
Using TensorFlow backend.
[INFO] loading pre-trained network...
[INFO] captcha: 2696
[INFO] captcha: 2337
[INFO] captcha: 2571
[INFO] captcha: 8648
```

In Figure 21.7 I have included four samples generated from my run of `test_model.py`. In *every case* we have correctly predicted the digit string and broken the image captcha using a simple network architecture trained on a small amount of training data.

21.2 Summary

In this chapter we learned how to:

1. Gather a dataset of raw images.
2. Label and annotate our images for training.
3. Train a a custom Convolutional Neural Network on our labeled dataset.
4. Test and evaluate our model on example images.

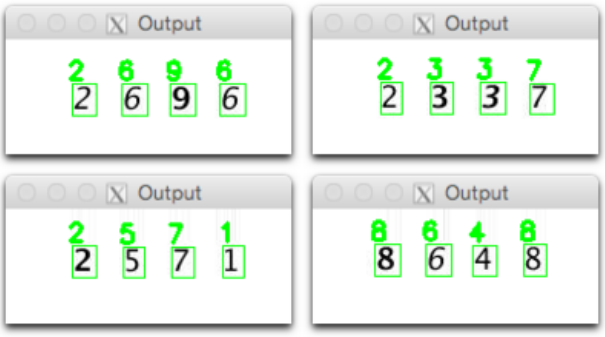

Figure 21.7: Examples of captchas that have been correctly classified and broken by our LeNet model.

To accomplish this, we scraped 500 example captcha images from the E-ZPass NY website. We then wrote a Python script that aids us in the labeling process, enabling us to quickly label the entire dataset and store the resulting images in an organized directory structure.

After our dataset was labeled, we trained the LeNet architecture using the SGD optimizer on the dataset using categorical crosse-entropy loss – the resulting model obtained 100% accuracy on the testing set with zero overfitting. Finally, we visualized results of the predicted digits to confirm that we have successfully devised a method to break the captcha.

Again, I want to remind you that this chapter serves as only an *example* of how to obtain an image dataset and label it. Under *no circumstances* should you use this dataset or resulting model for nefarious reasons. If you are ever in a situation where you find that computer vision or deep learning can be used to exploit a vulnerability, be sure to practice *responsible disclosure* and attempt to report the issue to the proper stakeholders; failure to do so is unethical (as is misuse of this code, which, legally, I must say I cannot take responsibility for).

Secondly, this chapter (as will the next one on smile detection with deep learning) have leveraged computer vision and the OpenCV library to facilitate building a complete application. If you are planning on becoming a serious deep learning practitioner, I *highly recommend* that you learn the fundamentals of image processing and the OpenCV library – having even a rudimentary understanding of these concepts will enable you to:

1. Appreciate deep learning at a higher level.
2. Develop more robust applications that use deep learning for image classification
3. Leverage image processing techniques to more quickly obtain your goals.

A great example of using basic image processing techniques to our advantage can be found in the Section 21.1.4 above where we were able to quickly annotate and label our dataset. Without using simple computer vision techniques, we would have been stuck manually cropping and saving the example digits to disk using image editing software such as Photoshop or GIMP. Instead, we were able to write a quick-and-dirty application that *automatically* extracted each digit from the captcha – all we had to do was press the proper key on our keyboard to label the image.

If you are new to the world of OpenCV or computer vision, or if you simply want to level up your skills, I would highly encourage you to work through my book, *Practical Python and OpenCV* [8]. The book is a quick read and will give you the foundation you need to be successful when applying deep learning to image classification and computer vision tasks.

The next chapter is from the *Practitioner Bundle*, a perfect fit if you want to study deep learning *in-depth*.

Take a look at the *"Competing in Kaggle: Dogs vs. Cats"* sample chapter below…

10. Competing in Kaggle: Dogs vs. Cats

In our previous chapter, we learned how to work with HDF5 and datasets too large to fit into memory. To do so, we defined a Python utility script that can be used to take an input dataset of images and serialize them into a highly efficient HDF5 dataset. Representing a dataset of images in an HDF5 dataset allows us to avoid issues of I/O latency, thereby speeding up the training process.

For example, if we defined a dataset generator that loaded images sequentially from disk, we would need N read operations, one for each image. However, by placing our dataset of images into an HDF5 dataset, we can instead load *batches* of images using a single read. This action *dramatically* reduces the number of I/O calls and allows us to work with very large image datasets.

In this chapter, we are going to extend our work and learn how to define an *image generator* for HDF5 datasets suitable for training Convolutional Neural Networks with Keras. This generator will open the HDF5 dataset, yield batches of images and associated training labels for the network to be trained on, and proceed to do so until our model reaches sufficiently low loss/high accuracy.

To accomplish this process, we'll first explore three new image pre-processors designed to increase classification accuracy – *mean subtraction*, *patch extraction*, and *cropping* (also called *10-cropping* or *over-sampling*). Once we've defined our new set of pre-processors, we'll move on defining the actual HDF5 dataset generator.

From there, we'll implement the seminal AlexNet architecture from Krizhevsky et al.'s 2012 paper, *ImageNet Classification with Deep Convolutional Neural Networks* [6]. This implementation of AlexNet will then be trained on the Kaggle Dogs vs. Cats challenge. Given the trained model, we'll evaluate its performance on the testing set, followed by using over-sampling methods to boost classification accuracy further. As our results will demonstrate, our network architecture + cropping methods will enable us to obtain a position in the top-25 leaderboard of the Kaggle Dogs vs. Cats challenge.

10.1 Additional Image Preprocessors

In this section we'll implement two new image pre-preprocessors:

1. A mean subtraction pre-processor designed to subtract the mean Red, Green, and Blue pixel intensities across a dataset from an input image (which is a form of data normalization).

2. A patch preprocessor used to randomly extract $M \times N$ pixel regions from an image during training.
3. An over-sampling pre-processor used at testing time to sample five regions of an input image (the four corners + center area) along with their corresponding horizontal flips (for a total of 10 crops).

Using over-sampling, we can boost our classification accuracy by passing the 10 crops through our CNN and then averaging across the 10 predictions.

10.1.1 Mean Preprocessing

Let's get started with the mean pre-processor. In Chapter 9 we learned how to convert an image dataset to HDF5 format – part of this conversion involved computing the average Red, Green, and Blue pixel intensities across *all images* in the entire dataset. Now that we have these averages, we are going to perform a pixel-wise subtraction of these values from our input images as a form of data normalization. Given an input image I and its R, G, B channels, we can perform mean subtraction via:

- $R = R - \mu_R$
- $G = G - \mu_G$
- $B = B - \mu_B$

Where μ_R, μ_G, and μ_B are computed when the image dataset is converted to HDF5 format. Figure 10.1 includes a visualization of subtracting the mean RGB values from an input image – notice how the subtraction is done pixel-wise.

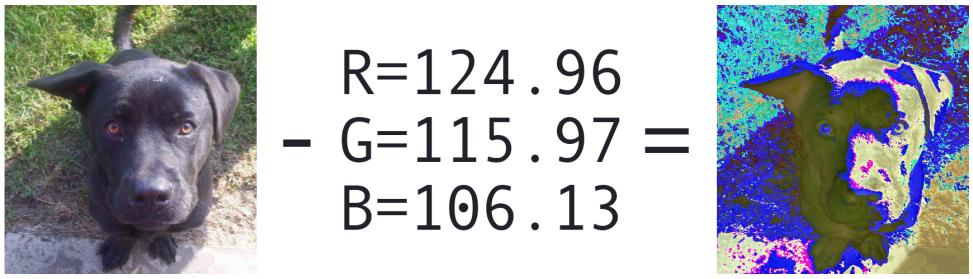

Figure 10.1: An example of applying mean subtraction to an input image (*left*) by subtracting $R = 124.96$, $G = 115.97$, $B = 106.13$ pixel-wise, resulting in the output image (*right*). Mean subtraction is used to reduce the affects of lighting variations during classification.

To make this concept more concrete, let's go ahead and implement our `MeanPreprocessor` class:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- nn
|    |--- preprocessing
|    |    |--- __init__.py
|    |    |--- aspectawarepreprocessor.py
|    |    |--- imagetoarraypreprocessor.py
|    |    |--- meanpreprocessor.py
|    |    |--- simplepreprocessor.py
|    |--- utils
```

Notice how I have placed a new file named `meanpreprocessor.py` in the `preprocessing` sub-module of `pyimagesearch` – this location is where our `MeanPreprocessor` class will live. Let's go ahead and implement this class now:

```
1  # import the necessary packages
2  import cv2
3
4  class MeanPreprocessor:
5      def __init__(self, rMean, gMean, bMean):
6          # store the Red, Green, and Blue channel averages across a
7          # training set
8          self.rMean = rMean
9          self.gMean = gMean
10         self.bMean = bMean
```

Line 5 defines the constructor to the `MeanPreprocessor`, which requires three arguments – the respective Red, Green, and Blue averages computed across the entire dataset. These values are then stored on **Lines 8-10**.

Next, let's define the `preprocess` method, a required function for *every* pre-processor we intend to apply to our image processing pipeline:

```
12     def preprocess(self, image):
13         # split the image into its respective Red, Green, and Blue
14         # channels
15         (B, G, R) = cv2.split(image.astype("float32"))
16
17         # subtract the means for each channel
18         R -= self.rMean
19         G -= self.gMean
20         B -= self.bMean
21
22         # merge the channels back together and return the image
23         return cv2.merge([B, G, R])
```

Line 15 uses the `cv2.split` function to split our input `image` into its respective RGB components. Keep in mind that OpenCV represents images in BGR order rather than RGB ([38], http://pyimg.co/ppao), hence why our return tuple has the signature `(B, G, R)` rather than `(R, G, B)`. We'll also ensure that these channels are of a floating point data type as OpenCV images are typically represented as unsigned 8-bit integers (in which case we can't have negative values, and modulo arithmetic would be performed instead).

Lines 17-20 perform the mean subtraction itself, subtracting the respective mean RGB values from the RGB channels of the input image. **Line 23** then merges the normalized channels back together and returns the resulting image to the calling function.

10.1.2 Patch Preprocessing

The `PatchPreprocessor` is responsible for randomly sampling $M \times N$ regions of an image during the *training process*. We apply patch preprocessing when the spatial dimensions of our input images are *larger* than what the CNN expects – this is a common technique to help reduce overfitting, and is, therefore, a form of regularization. Instead of using the *entire* image during training, we instead crop a random portion of it and pass it to the network (see Figure 10.2 for an example of crop preprocessing).

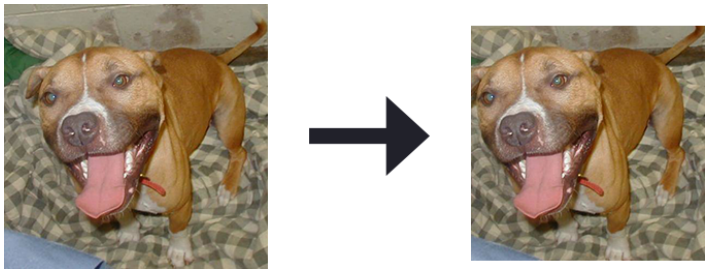

Figure 10.2: **Left:** Our original 256×256 input image. **Right:** *Randomly* cropping a 227×227 region from the image.

Applying this cropping implies that a network never sees the *exact* same image (unless by random happenstance), similar to data augmentation. As you know from our previous chapter, we constructed an HDF5 dataset of Kaggle Dogs vs. Cats images where each image is 256×256 pixels. However, the AlexNet architecture that we'll be implementing later in this chapter can only accept images of size 227×227 pixels.

So, what are we to do? Apply a `SimplePreprocessor` to resize our each of the 256×256 pixels down to 227×227? No, that would be wasteful, especially since this is an excellent opportunity to perform data augmentation by *randomly cropping* a 227×227 region from the 256×256 image during training – in fact, this process is exactly how Krizhevsky et al. trains AlexNet on the ImageNet dataset.

The `PatchPreprocessor`, just like all other image pre-processors, will be sorted in the `preprocessing` sub-module of `pyimagesearch`:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- nn
|    |--- preprocessing
|    |    |--- __init__.py
|    |    |--- aspectawarepreprocessor.py
|    |    |--- imagetoarraypreprocessor.py
|    |    |--- meanpreprocessor.py
|    |    |--- patchpreprocessor.py
|    |    |--- simplepreprocessor.py
|    |--- utils
```

Open up the `patchpreprocessor.py` file and let's define the `PatchPreprocessor` class:

```
1 # import the necessary packages
2 from sklearn.feature_extraction.image import extract_patches_2d
3
4 class PatchPreprocessor:
5     def __init__(self, width, height):
6         # store the target width and height of the image
7         self.width = width
8         self.height = height
```

Line 5 defines the construct to `PatchPreprocessor` – we simply need to supply the target `width` and `height` of the cropped image.

We can then define the `preprocess` function:

```
10      def preprocess(self, image):
11          # extract a random crop from the image with the target width
12          # and height
13          return extract_patches_2d(image, (self.height, self.width),
14              max_patches=1)[0]
```

Extracting a random patches of size `self.width x self.height` is easy using the `extract_patches_2d` function from the scikit-learn library. Given an input `image`, this function randomly extracts a patch from image. Here we supply `max_patches=1`, indicating that we only need a *single* random patch from the input image.

The `PatchPreprocessor` class doesn't seem like much, but it's actually a very effective method to avoid overfitting by applying yet another layer of data augmentation. We'll be using the `PatchPreprocessor` when training AlexNet. The next pre-processor, `CropPreprocessor`, will be used when evaluating our trained network.

10.1.3 Crop Preprocessing

Next, we need to define a `CropPreprocessor` responsible for computing the 10-crops for over-sampling. During the evaluating phase of our CNN, we'll crop the four corners of the input image + the center region and then take their corresponding horizontal flips, for a total of ten samples per input image (Figure 10.3).

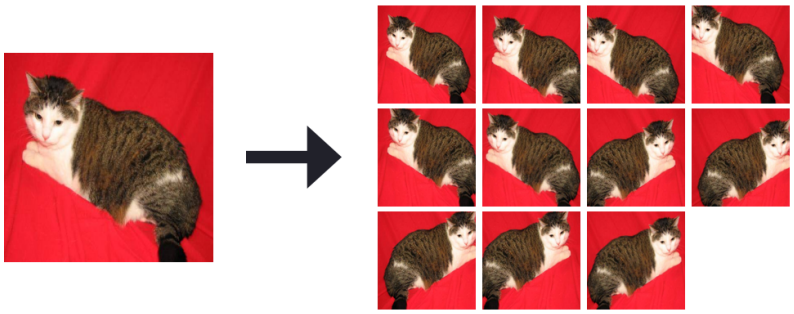

Figure 10.3: **Left:** The original 256×256 input image. **Right:** Applying the 10-crop preprocessor to extract ten 227×227 crops of the image including the center, four corners, and their corresponding horizontal mirrors.

These ten samples will be passed through the CNN, and then the probabilities averaged. Applying this over-sampling method tends to include 1-2 percent increases in classification accuracy (and in some cases, even higher).

The `CropPreprocessor` class will also live in the `preprocessing` sub-module of `pyimagesearch`:

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

```
|    |--- preprocessing
|    |    |--- __init__.py
|    |    |--- aspectawarepreprocessor.py
|    |    |--- croppreprocessor.py
|    |    |--- imagetoarraypreprocessor.py
|    |    |--- meanpreprocessor.py
|    |    |--- patchpreprocessor.py
|    |    |--- simplepreprocessor.py
|    |--- utils
```

Open up the `croppreprocessor.py` file and let's define it:

```
1  # import the necessary packages
2  import numpy as np
3  import cv2
4
5  class CropPreprocessor:
6       def __init__(self, width, height, horiz=True, inter=cv2.INTER_AREA):
7            # store the target image width, height, whether or not
8            # horizontal flips should be included, along with the
9            # interpolation method used when resizing
10           self.width = width
11           self.height = height
12           self.horiz = horiz
13           self.inter = inter
```

Line 6 defines the constructor to to `CropPreprocessor`. The only *required* arguments are the target `width` and `height` of each cropped region. We can also optionally specify whether horizontal flipping should be applied (defaults to `True`) along with the interpolation algorithm OpenCV will use for resizing. These arguments are all stored inside the class for use within the `preprocess` method.

Speaking of which, let's define the `preprocess` method now:

```
15      def preprocess(self, image):
16           # initialize the list of crops
17           crops = []
18
19           # grab the width and height of the image then use these
20           # dimensions to define the corners of the image based
21           (h, w) = image.shape[:2]
22           coords = [
23                [0, 0, self.width, self.height],
24                [w - self.width, 0, w, self.height],
25                [w - self.width, h - self.height, w, h],
26                [0, h - self.height, self.width, h]]
27
28           # compute the center crop of the image as well
29           dW = int(0.5 * (w - self.width))
30           dH = int(0.5 * (h - self.height))
31           coords.append([dW, dH, w - dW, h - dH])
```

The `preprocess` method requires only a single argument – the `image` which we are going to apply over-sampling. We grab the width and height of the input image on **Line 21**, which then

allows us to compute the (x, y)-coordinates of the four corners (top-left, top-right, bottom-right, bottom-left, respectively) on **Lines 22-26**. The center crop of the image is then computed on **Lines 29 and 30**, then added to the list of `coords` on **Line 31**.

We are now ready to extract each of the crops:

```
33          # loop over the coordinates, extract each of the crops,
34          # and resize each of them to a fixed size
35          for (startX, startY, endX, endY) in coords:
36               crop = image[startY:endY, startX:endX]
37               crop = cv2.resize(crop, (self.width, self.height),
38                    interpolation=self.inter)
39               crops.append(crop)
```

On **Line 35** we loop over each of the starting and ending (x, y)-coordinates of the rectangular crops. **Line 36** extracts the `crop` via NumPy array slicing which we then resize on **Line 37** to ensure the target `width` and `height` dimensions are met. The `crop` is the added to the `crops` list.

In the case that horizontal mirrors are to be computed, we can flip each of the five original crops, leaving us with ten crops overall:

```
41          # check to see if the horizontal flips should be taken
42          if self.horiz:
43               # compute the horizontal mirror flips for each crop
44               mirrors = [cv2.flip(c, 1) for c in crops]
45               crops.extend(mirrors)
46
47          # return the set of crops
48          return np.array(crops)
```

The array of `crops` is then returned to the calling function on **Line 48**. Using both the `MeanPreprocessor` for normalization and the `CropPreprocessor` for oversampling, we'll be able to obtain higher classification accuracy than is otherwise possible.

10.2 HDF5 Dataset Generators

Before we can implement the AlexNet architecture and train it on the Kaggle Dogs vs. Cats dataset, we first need to define a class responsible for yielding *batches* of images and labels from our HDF5 dataset. Chapter 9 discussed how to convert a set of images residing on disk into an HDF5 dataset – but how do we get them back out again?

The answer is to define an `HDF5DatasetGenerator` class in the `io` sub-module of `pyimagesearch`:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- io
|    |    |--- __init__.py
|    |    |--- hdf5datasetgenerator.py
|    |    |--- hdf5datasetwriter.py
|    |--- nn
|    |--- preprocessing
|    |--- utils
```

Previously, all of our image datasets could be loaded into memory so we could rely on Keras generator utilities to yield our batches of images and corresponding labels. However, now that our datasets are too large to fit into memory, we need to handle implementing this generator ourselves.

Go ahead and open the `hdf5datasetgenerator.py` file and we'll get to work:

```
1  # import the necessary packages
2  from keras.utils import np_utils
3  import numpy as np
4  import h5py
5
6  class HDF5DatasetGenerator:
7      def __init__(self, dbPath, batchSize, preprocessors=None,
8          aug=None, binarize=True, classes=2):
9          # store the batch size, preprocessors, and data augmentor,
10         # whether or not the labels should be binarized, along with
11         # the total number of classes
12         self.batchSize = batchSize
13         self.preprocessors = preprocessors
14         self.aug = aug
15         self.binarize = binarize
16         self.classes = classes
17
18         # open the HDF5 database for reading and determine the total
19         # number of entries in the database
20         self.db = h5py.File(dbPath)
21         self.numImages = self.db["labels"].shape[0]
```

On **Line 7** we define the constructor to our `HDF5DatasetGenerator`. This class accepts a number of arguments, two of which are required and the rest optional. I have detailed each of the arguments below:

- `dbPath`: The path to our HDF5 dataset that stores our images and corresponding class labels.
- `batchSize`: The size of mini-batches to yield when training our network.
- `preprocessors`: The list of image preprocessors we are going to apply (i.e., `MeanPreprocessor`, `ImageToArrayPreprocessor`, etc.).
- `aug`: Defaulting to `None`, we could also supply a Keras `ImageDataGenerator` to apply data augmentation *directly inside* our `HDF5DatasetGenerator`.
- `binarize`: Typically we will store class labels as *single integers* inside our HDF5 dataset; however, as we know, if we are applying categorical cross-entropy or binary cross-entropy as our loss function, we first need to *binarize* the labels as one-hot encoded vectors – this switch indicates whether or not this binarization needs to take place (which defaults to `True`).
- `classes`: The number of unique class labels in our dataset. This value is required to accurately construct our one-hot encoded vectors during the binarization phase.

These variables are stored on **Lines 12-16** so we can access them from the rest of the class. **Line 20** opens a file pointer to our HDF5 dataset file **Line 21** creates a convenience variable used to access the total number of data points in the dataset.

Next, we need to define a `generator` function, which as the name suggests, is responsible for yielding batches of images and class labels to the Keras `.fit_generator` function when training a network:

```
23     def generator(self, passes=np.inf):
24         # initialize the epoch count
```

```
25          epochs = 0
26
27          # keep looping infinitely -- the model will stop once we have
28          # reach the desired number of epochs
29          while epochs < passes:
30               # loop over the HDF5 dataset
31               for i in np.arange(0, self.numImages, self.batchSize):
32                    # extract the images and labels from the HDF dataset
33                    images = self.db["images"][i: i + self.batchSize]
34                    labels = self.db["labels"][i: i + self.batchSize]
```

Line 23 defines the `generator` function which can accept an optional argument, `passes`. Think of the `passes` value as the total number of *epochs* – in *most cases*, we don't want our generator to be concerned with the total number of epochs; our training methodology (fixed number of epochs, early stopping, etc.) should be responsible for that. However, in certain situations, it's often helpful to provide this information to the generator.

On **Line 29** we start looping over the number of desired epochs – by default, this loop will run indefinitely until either:

1. Keras reaches training termination criteria.
2. We explicitly stop the training process (i.e., ctrl + c).

Line 31 starts looping over each batch of data points in the dataset. We extract the `images` and `labels` of size `batchSize` from our HDF5 dataset on **Lines 33 and 34**.

Next, let's check to see if the `labels` should be one-hot encoded:

```
36                    # check to see if the labels should be binarized
37                    if self.binarize:
38                         labels = np_utils.to_categorical(labels,
39                              self.classes)
```

We can then also see if any image `preprocessors` should be applied:

```
41                    # check to see if our preprocessors are not None
42                    if self.preprocessors is not None:
43                         # initialize the list of processed images
44                         procImages = []
45
46                         # loop over the images
47                         for image in images:
48                              # loop over the preprocessors and apply each
49                              # to the image
50                              for p in self.preprocessors:
51                                   image = p.preprocess(image)
52
53                              # update the list of processed images
54                              procImages.append(image)
55
56                         # update the images array to be the processed
57                         # images
58                         images = np.array(procImages)
```

Provided the `preprocessors` is not `None` (**Line 42**), we loop over each of the images in the batch and apply each of the `preprocessors` by calling the `preprocess` method on the individual `image`. Doing this enables us to *chain together* multiple image pre-processors.

For example, our first pre-processor may resize the image to a fixed size via our `SimplePreprocessor` class. From there we may perform mean subtraction via the `MeanPreprocessor`. And after that, we'll need to convert the image to a Keras-compatible array using the `ImageToArrayPreprocessor`. At this point it should be clear why we defined all of our pre-processing classes with a `preprocess` method – it allows us to *chain* our pre-processors together inside the data generator. The preprocessed images are then converted back to a NumPy array on **Line 58**.

Provided we supplied an instance of `aug`, an `ImageDataGenerator` class used for data augmentation, we'll also want to apply data augmentation to the images as well:

```
60                      # if the data augmenator exists, apply it
61                      if self.aug is not None:
62                          (images, labels) = next(self.aug.flow(images,
63                              labels, batch_size=self.batchSize))
```

Finally, we can yield a 2-tuple of the batch of `images` and `labels` to the calling Keras generator:

```
65                      # yield a tuple of images and labels
66                      yield (images, labels)
67
68                  # increment the total number of epochs
69                  epochs += 1
70
71      def close(self):
72          # close the database
73          self.db.close()
```

Line 69 increments our total number of epochs after all mini-batches in the dataset have been processed. The `close` method on **Lines 71-73** is simply responsible for closing the pointer to the HDF5 dataset.

Admittedly, implementing the `HDF5DatasetGenerator` may not "feel" like we're doing any deep learning. After all, isn't this just a class responsible for yielding batches of data from a file? Technically, yes, that is correct. However, keep in mind that *practical* deep learning is more than just defining a model architecture, initializing an optimizer, and applying it to a dataset.

In reality, we need *extra tools* to help facilitate our ability to work with datasets, *especially* datasets that are too large to fit into memory. As we'll see throughout the rest of this book, our `HDF5DatasetGenerator` will come in handy a number of times – and when you start creating your own deep learning applications/experiments, you'll feel quite lucky to have it in your repertoire.

10.3 Implementing AlexNet

Let's now move on to implement the seminal AlexNet architecture by Krizhevsky et al. A table summarizing the AlexNet architecture can be seen in Table 10.1.

Notice how our input images are assumed to be $227 \times 227 \times 3$ pixels – this is actually the *correct* input size for AlexNet. As mentioned in Chapter 9, in the original publication, Krizhevsky et al. reported the input spatial dimensions to be $224 \times 224 \times 3$; however, since we know 224×224 cannot possible by tiled with an 11×1 kernel, we assume there was likely a typo in the publication, and 224×224 should actually be 227×227.

The first block of AlexNet applies 96, 11×11 kernels with a stride of 4×4, followed by a RELU activation and max pooling with a pool size of 3×3 and strides of 2×2, resulting in an output volume of size 55×55.

Layer Type	**Output Size**	**Filter Size / Stride**
INPUT IMAGE	$227 \times 227 \times 3$	
CONV	$57 \times 57 \times 96$	$11 \times 11/4 \times 4, K = 96$
ACT	$57 \times 57 \times 96$	
BN	$57 \times 57 \times 96$	
POOL	$16 \times 16 \times 96$	$3 \times 3/2 \times 2$
DROPOUT	$28 \times 28 \times 96$	
CONV	$28 \times 28 \times 256$	$5 \times 5, K = 256$
ACT	$28 \times 28 \times 256$	
BN	$28 \times 28 \times 256$	
POOL	$13 \times 13 \times 256$	$3 \times 3/2 \times 2$
DROPOUT	$13 \times 13 \times 256$	
CONV	$13 \times 13 \times 384$	$3 \times 3, K = 384$
ACT	$13 \times 13 \times 384$	
BN	$13 \times 13 \times 384$	
CONV	$13 \times 13 \times 384$	$3 \times 3, K = 384$
ACT	$13 \times 13 \times 384$	
BN	$13 \times 13 \times 384$	
CONV	$13 \times 13 \times 256$	$3 \times 3, K = 256$
ACT	$13 \times 13 \times 256$	
BN	$13 \times 13 \times 256$	
POOL	$13 \times 13 \times 256$	$3 \times 3/2 \times 2$
DROPOUT	$6 \times 6 \times 256$	
FC	4096	
ACT	4096	
BN	4096	
DROPOUT	4096	
FC	4096	
ACT	4096	
BN	4096	
DROPOUT	4096	
FC	1000	
SOFTMAX	1000	

Table 10.1: A table summary of the AlexNet architecture. Output volume sizes are included for each layer, along with convolutional filter size/pool size when relevant.

We then apply a second `CONV => RELU => POOL` layer this, this time using 256, 5×5 filters with 1×1 strides. After applying max pooling again with a pool size of 3×3 and strides of 2×2 we are left with a 13×13 volume.

Next, we apply `(CONV => RELU) * 3 => POOL`. The first two `CONV` layers learn 384, 3×3 filters while the final `CONV` learns 256, 3×3 filters.

After another max pooling operation, we reach our two `FC` layers, each with 4096 nodes and RELU activations in between. The final layer in the network is our softmax classifier.

When AlexNet was first introduced we did not have techniques such as batch normalization – in our implementation ,we are going to include batch normalization after the activation, as is standard for the majority of image classification tasks using Convolutional Neural Networks. We'll also include a very small amount of dropout after each `POOL` operation to further help reduce overfitting.

To implement AlexNet, let's create a new file named `alexnet.py` in the `conv` sub-module of `nn` in `pyimagesearch`:

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

From there, open up `alexnet.py`, and we'll implement this seminal architecture:

```
1  # import the necessary packages
2  from keras.models import Sequential
3  from keras.layers.normalization import BatchNormalization
4  from keras.layers.convolutional import Conv2D
5  from keras.layers.convolutional import MaxPooling2D
6  from keras.layers.core import Activation
7  from keras.layers.core import Flatten
8  from keras.layers.core import Dropout
9  from keras.layers.core import Dense
10 from keras.regularizers import l2
11 from keras import backend as K
```

Lines 2-11 import our required Keras classes – we have used all of these layers before in previous chapters of this book so I'm going to skip explicitly describing each of them. The only import I *do* want to draw your attention to is **Line 10** where we import the `l2` function – this method will be responsible for applying L2 weight decay to the weight layers in the network.

Now that our imports are taken care of, let's start the definition of AlexNet:

```
13 class AlexNet:
14     @staticmethod
15     def build(width, height, depth, classes, reg=0.0002):
16         # initialize the model along with the input shape to be
```

```
17          # "channels last" and the channels dimension itself
18          model = Sequential()
19          inputShape = (height, width, depth)
20          chanDim = -1
21
22          # if we are using "channels first", update the input shape
23          # and channels dimension
24          if K.image_data_format() == "channels_first":
25               inputShape = (depth, height, width)
26               chanDim = 1
```

Line 15 defines the `build` method of `AlexNet`. Just like in all previous examples in this book, the `build` method is required for constructing the actual network architecture and returning it to the calling function. This method accepts four arguments: the `width`, `height`, and `depth` of the input images, followed by the total number of `class` labels in the dataset. An optional parameter, `reg`, controls the amount of L2 regularization we'll be applying to the network. For larger, deeper networks, applying regularization is *critical* to reducing overfitting while increasing accuracy on the validation and testing sets.

Line 18 initializes the `model` itself along with the `inputShape` and channel dimension assuming we are using "channels last" ordering. If we are instead using "channels first" ordering, we update `inputShape` and `chanDim` (**Lines 24-26**).

Let's now define the first `CONV => RELU => POOL` layer set in the network:

```
28          # Block #1: first CONV => RELU => POOL layer set
29          model.add(Conv2D(96, (11, 11), strides=(4, 4),
30               input_shape=inputShape, padding="same",
31               kernel_regularizer=l2(reg)))
32          model.add(Activation("relu"))
33          model.add(BatchNormalization(axis=chanDim))
34          model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))
35          model.add(Dropout(0.25))
```

Our first `CONV` layer will learn 96 filters, each of size 11×11 (**Lines 28 and 29**), using a stride of 4×4. By applying the `kernel_regularizer` parameter to the `Conv2D` class, we can apply our L2 weight regularization parameter – this regularization will be applied to *all* `CONV` and `FC` layers in the network.

A ReLU activation is applied after our `CONV`, followed by a `BatchNormalization` (**Lines 32 and 33**). The `MaxPooling2D` is then applied to reduce our spatial dimensions (**Line 34**). We'll also apply dropout with a small probability (25 percent) to help reduce overfitting (**Lines 35**).

The following code block defines another `CONV => RELU => POOL` layer set, this time learning 256 filters, each of size 5×5:

```
37          # Block #2: second CONV => RELU => POOL layer set
38          model.add(Conv2D(256, (5, 5), padding="same",
39               kernel_regularizer=l2(reg)))
40          model.add(Activation("relu"))
41          model.add(BatchNormalization(axis=chanDim))
42          model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))
43          model.add(Dropout(0.25))
```

Deeper, richer features are learned in the third block of AlexNet where we stack *multiple* `CONV => RELU` together prior to applying a `POOL` operation:

```
45          # Block #3: CONV => RELU => CONV => RELU => CONV => RELU
46          model.add(Conv2D(384, (3, 3), padding="same",
47               kernel_regularizer=l2(reg)))
48          model.add(Activation("relu"))
49          model.add(BatchNormalization(axis=chanDim))
50          model.add(Conv2D(384, (3, 3), padding="same",
51               kernel_regularizer=l2(reg)))
52          model.add(Activation("relu"))
53          model.add(BatchNormalization(axis=chanDim))
54          model.add(Conv2D(256, (3, 3), padding="same",
55               kernel_regularizer=l2(reg)))
56          model.add(Activation("relu"))
57          model.add(BatchNormalization(axis=chanDim))
58          model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))
59          model.add(Dropout(0.25))
```

The first two CONV filters learn 384, 3×3 filters while the third CONV learns 256, 3×3 filters. Again, stacking multiple CONV =>RELU layers on to of each other *prior* to applying a destructive POOL layer enables our network to learn richer, and potentially more discriminating features.

From there we collapse our multi-dimensional representation down into a standard feedforward network using two fully-connected layers (4096 nodes each):

```
61          # Block #4: first set of FC => RELU layers
62          model.add(Flatten())
63          model.add(Dense(4096, kernel_regularizer=l2(reg)))
64          model.add(Activation("relu"))
65          model.add(BatchNormalization())
66          model.add(Dropout(0.5))
67
68          # Block #5: second set of FC => RELU layers
69          model.add(Dense(4096, kernel_regularizer=l2(reg)))
70          model.add(Activation("relu"))
71          model.add(BatchNormalization())
72          model.add(Dropout(0.5))
```

Batch normalization is applied after each activation in the FC layer sets, just as in the CONV layers above. Dropout, with a larger probability of 50 percent, is applied after every FC layer set, as is standard with the vast majority of CNNs.

Finally, we define the softmax classifier using the desired number of classes and return the resulting model to the calling function:

```
74          # softmax classifier
75          model.add(Dense(classes, kernel_regularizer=l2(reg)))
76          model.add(Activation("softmax"))
77
78          # return the constructed network architecture
79          return model
```

As you can see, implementing AlexNet is a fairly straightforward process, *especially* when you have the "blueprint" of the architecture presented in Table 10.1 above. Whenever implementing architectures from publications, try to see if they provide such a table as it makes implementation

much easier. For your own network architectures, use Chapter 19 of the *Starter Bundle* on visualizing network architectures to aid you in ensuring your input volume and output volume sizes are what you expect.

10.4 Training AlexNet on Kaggle: Dogs vs. Cats

Now that the AlexNet architecture has been defined, let's apply it to the Kaggle Dogs vs. Cats challenge. Open up a new file, name it `train_alexnet.py`, and insert the following code:

```
1  # import the necessary packages
2  # set the matplotlib backend so figures can be saved in the background
3  import matplotlib
4  matplotlib.use("Agg")
5
6  # import the necessary packages
7  from config import dogs_vs_cats_config as config
8  from pyimagesearch.preprocessing import ImageToArrayPreprocessor
9  from pyimagesearch.preprocessing import SimplePreprocessor
10 from pyimagesearch.preprocessing import PatchPreprocessor
11 from pyimagesearch.preprocessing import MeanPreprocessor
12 from pyimagesearch.callbacks import TrainingMonitor
13 from pyimagesearch.io import HDF5DatasetGenerator
14 from pyimagesearch.nn.conv import AlexNet
15 from keras.preprocessing.image import ImageDataGenerator
16 from keras.optimizers import Adam
17 import json
18 import os
```

Lines 3 and 4 import matplotlib, while ensuring the backend is set such that we can save figures and plots to disk as our network trains. We then implement our pre-processors on **Lines 8-11**. The `HDF5DatasetGenerator` is then imported on **Line 13** so we can access batches of training data from our serialized HDF5 dataset. AlexNet is also implement don **Line 14**.

Our next code block handles initializing our data augmentation generator via the `ImageDataGenerator` class:

```
20 # construct the training image generator for data augmentation
21 aug = ImageDataGenerator(rotation_range=20, zoom_range=0.15,
22     width_shift_range=0.2, height_shift_range=0.2, shear_range=0.15,
23     horizontal_flip=True, fill_mode="nearest")
```

Let's take the time now to initialize each of our image pre-processors:

```
25 # load the RGB means for the training set
26 means = json.loads(open(config.DATASET_MEAN).read())
27
28 # initialize the image preprocessors
29 sp = SimplePreprocessor(227, 227)
30 pp = PatchPreprocessor(227, 227)
31 mp = MeanPreprocessor(means["R"], means["G"], means["B"])
32 iap = ImageToArrayPreprocessor()
```

On **Line 26** we load the serialized RGB means from disk – these are the means for each of the respective Red, Green, and Blue channels across our training dataset. These values will later be passed into a `MeanPreprocessor` for mean subtraction normalization.

Line 29 instantiates a `SimplePreprocessor` used to resize an input image down to 227×227 pixels. This pre-processor will be used in the *validation* data generator as our input images are 256×256 pixels; however, AlexNet is intended to handle only 227×227 images (hence why we need to resize the image during validation).

Line 30 instantiates a `PatchPreprocessor` – this pre-processor will randomly sample 227×227 regions from the 256×256 input images during training time, serving as a second form of data augmentation.

We then initialize the `MeanPreprocessor` on **Line 31** using our respective, Red, Green, and Blue averages. Finally, the `ImageToArrayPreprocessor` (**Line 32**) is used to convert images to Keras-compatible arrays.

Given our pre-processors, let's define the `HDF5DatasteGenerator` for both the training and validation data:

```
34 # initialize the training and validation dataset generators
35 trainGen = HDF5DatasetGenerator(config.TRAIN_HDF5, 128, aug=aug,
36     preprocessors=[pp, mp, iap], classes=2)
37 valGen = HDF5DatasetGenerator(config.VAL_HDF5, 128,
38     preprocessors=[sp, mp, iap], classes=2)
```

Lines 35 and 36 create our *training* dataset generator. Here we supply the path to our training HDF5 file, indicating that we should use batch sizes of 128 images, data augmentation, and three pre-processors: patch, mean, and image to array, respectively.

Lines 37 and 38 are responsible for instantiating the *testing* generator. This time we'll supply the path to the validation HDF5 file, use a batch size of 128, *no data augmentation*, and a simple pre-processor rather than a patch pre-processor (since data augmentation is not applied to validation data).

Finally, we are ready to initialize the Adam optimizer and AlexNet architecture:

```
40 # initialize the optimizer
41 print("[INFO] compiling model...")
42 opt = Adam(lr=1e-3)
43 model = AlexNet.build(width=227, height=227, depth=3,
44     classes=2, reg=0.0002)
45 model.compile(loss="binary_crossentropy", optimizer=opt,
46     metrics=["accuracy"])
47
48 # construct the set of callbacks
49 path = os.path.sep.join([config.OUTPUT_PATH, "{}.png".format(
50     os.getpid())])
51 callbacks = [TrainingMonitor(path)]
```

On **Line 42** we instantiate the Adam optimizer using the default learning rate of 0.001. The reason I choose Adam for this experiment (rather than SGD) is two-fold:

1. I wanted to give you exposure to using the more advanced optimizers we covered in Chapter 7.
2. Adam performs better on this classification task than SGD (which I know from the multiple previous experiments I ran before publishing this book).

We then initialize AlexNet on **Lines 43 and 44**, indicating that each input image will have a width of 227 pixels, a height of 227 pixels, 3 channels, and the dataset itself will have two classes (one for dogs, and another for cats). We'll also apply a small regularization penalty of `0.0002` to help combat overfitting and increase the ability of our model to generalize to the testing set.

We'll use *binary* cross-entropy rather than *categorical* cross-entropy (**Lines 45 and 46**) as this is only a two-class classification problem. We'll also define a `TrainingMonitor` callback on **Line 51** so we can monitor the performance of our network as it trains.

Speaking of training the network, let's do that now:

```
53 # train the network
54 model.fit_generator(
55     trainGen.generator(),
56     steps_per_epoch=trainGen.numImages // 128,
57     validation_data=valGen.generator(),
58     validation_steps=valGen.numImages // 128,
59     epochs=75,
60     max_queue_size=128 * 2,
61     callbacks=callbacks, verbose=1)
```

To train AlexNet on the Kaggle Dogs vs. Cats dataset using our `HDF5DatasetGenerator`, we need to use the `fit_generator` method of the `model`. First, we pass in `trainGen.generator()`, the HDF5 generator used to construct mini-batches of training data (**Line 55**). To determine the number of batches per epoch, we divide the total number of images in the training set by our batch size (**Line 56**). We do the same on **Lines 57 and 58** for the validation data. Finally, we'll indicate that AlexNet is to be trained for 75 epochs.

The last step is to simply serialize our `model` to file after training, along with closing each of the training and testing HDF5 datasets, respectively:

```
63 # save the model to file
64 print("[INFO] serializing model...")
65 model.save(config.MODEL_PATH, overwrite=True)
66
67 # close the HDF5 datasets
68 trainGen.close()
69 valGen.close()
```

To train AlexNet on the Kaggle Dogs vs. Cats dataset, execute the following command:

```
$ python train_alexnet.py
Epoch 73/75
415s - loss: 0.4862 - acc: 0.9126 - val_loss: 0.6826 - val_acc: 0.8602
Epoch 74/75
408s - loss: 0.4865 - acc: 0.9166 - val_loss: 0.6894 - val_acc: 0.8721
Epoch 75/75
401s - loss: 0.4813 - acc: 0.9166 - val_loss: 0.4195 - val_acc: 0.9297
[INFO] serializing model...
```

A plot of the training and validation loss/accuracy over the 75 epochs can be seen in Figure 10.4. Overall we can see that the training and accuracy plots correlate well with each other, although we could help stabilize variations in validation loss towards the end of the 75 epoch cycle by applying

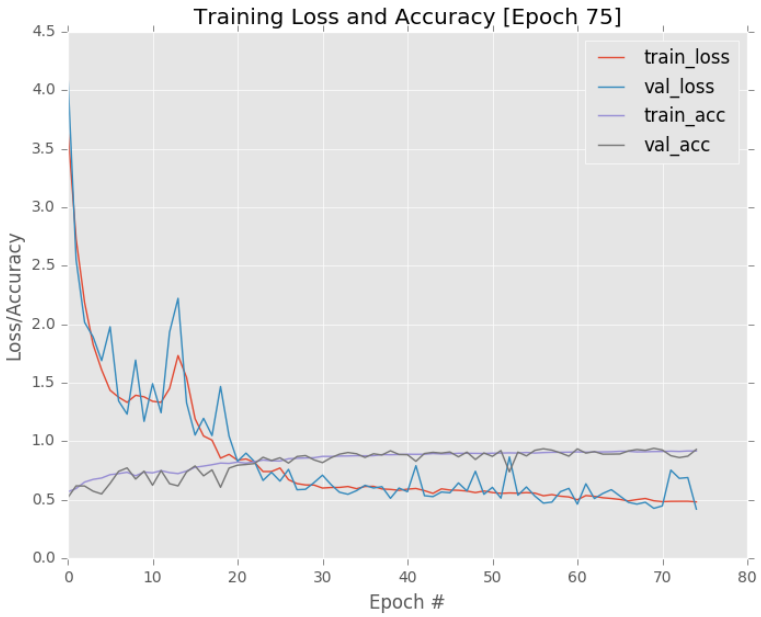

Figure 10.4: Training AlexNet on the Kaggle Dogs vs. Cats competition where we obtain 92.97% classification accuracy on our validation set. Our learning curve is stable with changes in training accuracy/loss being reflected in the respective validation split.

a bit of learning rate decay. Examining the classification report of AlexNet on the Dogs vs. Cats dataset, we see our obtained obtained **92.97**% on the validation set.

In the next section, we'll evaluate AlexNet on the testing set using both the standard method and over-sampling method. As our results will demonstrate, using over-sampling can increase your classification from 1-3% depending on your dataset and network architecture.

10.5 Evaluating AlexNet

To evaluate AlexNet on the testing set using both our standard method and over-sampling technique, let's create a new file named `crop_accuracy.py`:

```
--- dogs_vs_cats
|    |--- config
|    |--- build_dogs_vs_cats.py
|    |--- crop_accuracy.py
|    |--- extract_features.py
|    |--- train_alexnet.py
|    |--- train_model.py
|    |--- output
```

From there, open `crop_accuracy.py` and insert the following code:

```
1 # import the necessary packages
2 from config import dogs_vs_cats_config as config
```

```
3  from pyimagesearch.preprocessing import ImageToArrayPreprocessor
4  from pyimagesearch.preprocessing import SimplePreprocessor
5  from pyimagesearch.preprocessing import MeanPreprocessor
6  from pyimagesearch.preprocessing import CropPreprocessor
7  from pyimagesearch.io import HDF5DatasetGenerator
8  from pyimagesearch.utils.ranked import rank5_accuracy
9  from keras.models import load_model
10 import numpy as np
11 import progressbar
12 import json
```

Lines 2-12 import our required Python packages. **Line 2** imports our Python configuration file for the Dogs vs. Cats challenge. We'll also import our image preprocessors on **Lines 3-6**, including the `ImageToArrayPreprocessor`, `SimplePreprocessor`, `MeanPreprocessor`, and `CropPreprocessor`. The `HDF5DatasetGenerator` is required so we can access the *testing set* of our dataset and obtain predictions on this data using our pre-trained model.

Now that our imports are complete, let's load the RGB means from disk, initialize our image pre-preprocessors, and load the pre-trained AlexNet network:

```
14 # load the RGB means for the training set
15 means = json.loads(open(config.DATASET_MEAN).read())
16
17 # initialize the image preprocessors
18 sp = SimplePreprocessor(227, 227)
19 mp = MeanPreprocessor(means["R"], means["G"], means["B"])
20 cp = CropPreprocessor(227, 227)
21 iap = ImageToArrayPreprocessor()
22
23 # load the pretrained network
24 print("[INFO] loading model...")
25 model = load_model(config.MODEL_PATH)
```

Before we apply over-sampling and 10-cropping, let's first obtain a baseline on the testing set using only the original testing image as input to our network:

```
27 # initialize the testing dataset generator, then make predictions on
28 # the testing data
29 print("[INFO] predicting on test data (no crops)...")
30 testGen = HDF5DatasetGenerator(config.TEST_HDF5, 64,
31      preprocessors=[sp, mp, iap], classes=2)
32 predictions = model.predict_generator(testGen.generator(),
33      steps=testGen.numImages // 64, max_queue_size=64 * 2)
34
35 # compute the rank-1 and rank-5 accuracies
36 (rank1, _) = rank5_accuracy(predictions, testGen.db["labels"])
37 print("[INFO] rank-1: {:.2f}%".format(rank1 * 100))
38 testGen.close()
```

Lines 30 and 31 initialize the `HDF5DatasetGenerator` to access the testing dataset in batches of 64 images. Since we are obtaining a *baseline*, we'll use only the `SimplePreprocessor` to resize the 256×256 input images down to 227×227 pixels, followed by mean normalization and

converting the batch to a Keras-compatible array of images. **Lines 32 and 33** then use the generator to evaluate AlexNet on the dataset.

Given our `predictions`, we can compute our accuracy on the test set (**Lines 36-38**). Notice here how we only care about the `rank1` accuracy, which is because the Dogs vs. Cats is a 2-class dataset – computing the rank-5 accuracy for a 2-class dataset would trivially report 100 percent classification accuracy.

Now that we have a baseline for the standard evaluation technique, let's move on to over-sampling:

```
40 # re-initialize the testing set generator, this time excluding the
41 # `SimplePreprocessor`
42 testGen = HDF5DatasetGenerator(config.TEST_HDF5, 64,
43      preprocessors=[mp], classes=2)
44 predictions = []
45
46 # initialize the progress bar
47 widgets = ["Evaluating: ", progressbar.Percentage(), " ",
48      progressbar.Bar(), " ", progressbar.ETA()]
49 pbar = progressbar.ProgressBar(maxval=testGen.numImages // 64,
50      widgets=widgets).start()
```

On **Lines 42 and 43** we *re-initialize* the `HDF5DatasetGenerator`, this time instructing it to use *just* the `MeanPreprocessor` – we'll apply both over-sampling and Keras-array conversion later in the pipeline. **Lines 47-50** also initialize `progressbar` widgets to our screen if we are interested in having the evaluating progress displayed to our screen.

Given the re-instantiated `testGen`, we are now ready to apply the 10-cropping technique:

```
52 # loop over a single pass of the test data
53 for (i, (images, labels)) in enumerate(testGen.generator(passes=1)):
54      # loop over each of the individual images
55      for image in images:
56           # apply the crop preprocessor to the image to generate 10
57           # separate crops, then convert them from images to arrays
58           crops = cp.preprocess(image)
59           crops = np.array([iap.preprocess(c) for c in crops],
60                dtype="float32")
61
62           # make predictions on the crops and then average them
63           # together to obtain the final prediction
64           pred = model.predict(crops)
65           predictions.append(pred.mean(axis=0))
66
67      # update the progress bar
68      pbar.update(i)
```

On **Line 53** we start looping over every batch of images in the testing generator. Typically an `HDF5DatasetGenerator` is set to loop forever until we explicitly tell it to stop (normally by setting a maximum number of iterations via Keras when training); however, since we are now *evaluating*, we can supply `passes=1` to indicate the testing data only needs to be looped over once.

Then, for each image in the `images` batch (**Line 55**), we apply the 10-crop pre-processor on **Line 58**, which converts the `image` into an array of ten 227×227 images. These 227×227 crops were extracted from the original 256×256 batch based on the:

- Top-left corner
- Top-right corner
- Bottom-right corner
- Bottom-left corner
- Corresponding horizontal flips

Once we have the `crops`, we pass them through the `model` on **Line 64** for prediction. The final prediction (**Line 65**) is the *average of the probabilities* across all ten crops.

Our final code block handles displaying the accuracy of the over-sampling method:

```
70 # compute the rank-1 accuracy
71 pbar.finish()
72 print("[INFO] predicting on test data (with crops)...")
73 (rank1, _) = rank5_accuracy(predictions, testGen.db["labels"])
74 print("[INFO] rank-1: {:.2f}%".format(rank1 * 100))
75 testGen.close()
```

To evaluate AlexNet on the Kaggle Dog vs. Cats dataset, just execute the following command:

```
$ python crop_accuracy.py
[INFO] loading model...
[INFO] predicting on test data (no crops)...
[INFO] rank-1: 92.60%
Evaluating: 100% |####################################| Time: 0:01:12
[INFO] predicting on test data (with crops)...
[INFO] rank-1: 94.00%
```

As our results demonstrate, we reach **92.60%** accuracy on the testing set. However, by applying the 10-crop over-sampling method, we are able to boost classification accuracy to **94.00%**, an increase of 1.4%, which this was all accomplished simply by taking multiple crops of the input image and averaging the results. This straightforward, uncomplicated trick is an easy way to eke out an extra few percentage points when evaluating your network.

10.6 Obtaining a Top-5 Spot on the Kaggle Leaderboard

Of course, if you were to look at the Kaggle Dogs vs. Cats leaderboard, you would notice that to even break into the top-25 position we would need 96.69% accuracy, which our current method is not capable of reaching. So, what's the solution?

The answer is *transfer learning*, specifically transfer learning via feature extraction. While the ImageNet dataset consists of 1,000 object categories, a good portion of those include both *dog species* and *cat species*. Therefore, a network trained on ImageNet could not only tell you if an image was of a *dog* or a *cat*, but what particular *breed* the animal is as well. Given that a network trained on ImageNet must be able to discriminate between such fine-grained animals, it's natural to hypothesize that the features extracted from a pre-trained network would likely lend itself well to claiming a top spot on the Kaggle Dogs vs. Cats leaderboard.

To test this hypothesis, let's first extract features from the pre-trained ResNet architecture and then train a Logistic Regression classifier on top of these features.

10.6.1 Extracting Features Using ResNet

The transfer learning via feature extraction technique we'll be using in this section is heavily based on Chapter 3. I'll review the entire contents of `extract_features.py` as a matter of completeness;

however, please refer to Chapter 3 if you require further knowledge on feature extraction using CNNs.

To get started, open up a new file, name it `extract_features.py`, and insert the following code:

```
1  # import the necessary packages
2  from keras.applications import ResNet50
3  from keras.applications import imagenet_utils
4  from keras.preprocessing.image import img_to_array
5  from keras.preprocessing.image import load_img
6  from sklearn.preprocessing import LabelEncoder
7  from pyimagesearch.io import HDF5DatasetWriter
8  from imutils import paths
9  import numpy as np
10 import progressbar
11 import argparse
12 import random
13 import os
```

Lines 2-13 import our required Python packages. We import the `ResNet50` class on **Line 2** so we can access the pre-trained ResNet architecture. We'll also use the `HDF5DatasetWriter` on **Line 7** so we can write the extracted features to an efficiently HDF5 file format.

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

```
15 # construct the argument parse and parse the arguments
16 ap = argparse.ArgumentParser()
17 ap.add_argument("-d", "--dataset", required=True,
18      help="path to input dataset")
19 ap.add_argument("-o", "--output", required=True,
20      help="path to output HDF5 file")
21 ap.add_argument("-b", "--batch-size", type=int, default=16,
22      help="batch size of images to be passed through network")
23 ap.add_argument("-s", "--buffer-size", type=int, default=1000,
24      help="size of feature extraction buffer")
25 args = vars(ap.parse_args())
26
27 # store the batch size in a convenience variable
28 bs = args["batch_size"]
```

We only need two required command line arguments here, `--dataset`, which is the path to the input dataset of Dogs vs. Cats images, along with `--output`, the path to the output HDF5 file containing the features extracted via ResNet.

Next, let's grab the paths to the Dogs vs. Cats images residing on disk and then use the file paths to extract the label names:

```
30 # grab the list of images that we'll be describing then randomly
31 # shuffle them to allow for easy training and testing splits via
32 # array slicing during training time
33 print("[INFO] loading images...")
34 imagePaths = list(paths.list_images(args["dataset"]))
35 random.shuffle(imagePaths)
```

```
36
37 # extract the class labels from the image paths then encode the
38 # labels
39 labels = [p.split(os.path.sep)[2].split(".")[0] for p in imagePaths]
40 le = LabelEncoder()
41 labels = le.fit_transform(labels)
```

Now we can load our pre-trained ResNet50 weights from disk (excluding the FC layers):

```
43 # load the ResNet50 network
44 print("[INFO] loading network...")
45 model = ResNet50(weights="imagenet", include_top=False)
```

In order to store the features extracted from ResNet50 to disk, we need to instantiate a HDF5DatasetWriter object:

```
47 # initialize the HDF5 dataset writer, then store the class label
48 # names in the dataset
49 dataset = HDF5DatasetWriter((len(imagePaths), 2048),
50      args["output"], dataKey="features", bufSize=args["buffer_size"])
51 dataset.storeClassLabels(le.classes_)
```

The final average pooling layer of ResNet50 is 2048-d, hence why we supply a value of 2048 as the dimensionality to our HDF5datasetWriter.

We'll also initialize a progressbar so we can keep track of the feature extraction process:

```
53 # initialize the progress bar
54 widgets = ["Extracting Features: ", progressbar.Percentage(), " ",
55      progressbar.Bar(), " ", progressbar.ETA()]
56 pbar = progressbar.ProgressBar(maxval=len(imagePaths),
57      widgets=widgets).start()
```

Extracting features from a dataset using a CNN is the same as it was in Chapter 3. First, we loop over the imagePaths in batches:

```
59 # loop over the images in batches
60 for i in np.arange(0, len(imagePaths), bs):
61      # extract the batch of images and labels, then initialize the
62      # list of actual images that will be passed through the network
63      # for feature extraction
64      batchPaths = imagePaths[i:i + bs]
65      batchLabels = labels[i:i + bs]
66      batchImages = []
```

Followed by pre-processing each image:

```
68      # loop over the images and labels in the current batch
69      for (j, imagePath) in enumerate(batchPaths):
70           # load the input image using the Keras helper utility
```

```
71            # while ensuring the image is resized to 224x224 pixels
72            image = load_img(imagePath, target_size=(224, 224))
73            image = img_to_array(image)
74
75            # preprocess the image by (1) expanding the dimensions and
76            # (2) subtracting the mean RGB pixel intensity from the
77            # ImageNet dataset
78            image = np.expand_dims(image, axis=0)
79            image = imagenet_utils.preprocess_input(image)
80
81            # add the image to the batch
82            batchImages.append(image)
```

And then passing the `batchImages` through the network architecture, enabling us to extract features from the final `POOL` layer of ResNet50:

```
84      # pass the images through the network and use the outputs as
85      # our actual features
86      batchImages = np.vstack(batchImages)
87      features = model.predict(batchImages, batch_size=bs)
88
89      # reshape the features so that each image is represented by
90      # a flattened feature vector of the `MaxPooling2D` outputs
91      features = features.reshape((features.shape[0], 2048))
```

These extracted `features` are then added to our `dataset`:

```
93      # add the features and labels to our HDF5 dataset
94      dataset.add(features, batchLabels)
95      pbar.update(i)
96
97  # close the dataset
98  dataset.close()
99  pbar.finish()
```

To utilize ResNet to extract features from the Dogs vs. Cats dataset, simply execute the following command:

```
$ python extract_features.py --dataset ../datasets/kaggle_dogs_vs_cats/train \
      --output ../datasets/kaggle_dogs_vs_cats/hdf5/features.hdf5
[INFO] loading images...
[INFO] loading network...
Extracting Features: 100% |#####################################| Time: 0:06:18
```

After the command finishes executing, you should now have a file named `dogs_vs_cats_features.hdf5` in your `output` directory:

```
$ ls -l output/dogs_vs_cats_features.hdf5
-rw-rw-r-- adrian 409806272 Jun  3 07:17 output/dogs_vs_cats_features.hdf5
```

Given these features, we can train a Logistic Regression classier on top of them to (ideally) obtain a top-5 spot on the Kaggle Dogs vs. Cats leaderboard.

10.6.2 Training a Logistic Regression Classifier

To train our Logistic Regression classifier, open up a new file and name it `train_model.py`. From there, we can get started:

```
1  # import the necessary packages
2  from sklearn.linear_model import LogisticRegression
3  from sklearn.model_selection import GridSearchCV
4  from sklearn.metrics import classification_report
5  from sklearn.metrics import accuracy_score
6  import argparse
7  import pickle
8  import h5py
```

Lines 2-8 import our required Python packages. We'll then parse our command line arguments:

```
10 # construct the argument parse and parse the arguments
11 ap = argparse.ArgumentParser()
12 ap.add_argument("-d", "--db", required=True,
13      help="path HDF5 database")
14 ap.add_argument("-m", "--model", required=True,
15      help="path to output model")
16 ap.add_argument("-j", "--jobs", type=int, default=-1,
17      help="# of jobs to run when tuning hyperparameters")
18 args = vars(ap.parse_args())
```

We only need two switches here, the path to the input HDF5 `--db`, along with the path to the output Logistic Regression `--model` after training is complete.

Next, let's open the HDF5 dataset for reading and determine the training and testing split – 75% of the data for training and 25% for testing:

```
20 # open the HDF5 database for reading then determine the index of
21 # the training and testing split, provided that this data was
22 # already shuffled *prior* to writing it to disk
23 db = h5py.File(args["db"], "r")
24 i = int(db["labels"].shape[0] * 0.75)
```

Given this feature split, we'll perform a grid search over the `C` hyperparameter of the `LogisticRegression` classifier:

```
26 # define the set of parameters that we want to tune then start a
27 # grid search where we evaluate our model for each value of C
28 print("[INFO] tuning hyperparameters...")
29 params = {"C": [0.0001, 0.001, 0.01, 0.1, 1.0]}
30 model = GridSearchCV(LogisticRegression(), params, cv=3,
31      n_jobs=args["jobs"])
32 model.fit(db["features"][:i], db["labels"][:i])
33 print("[INFO] best hyperparameters: {}".format(model.best_params_))
```

Once we've found the best choice of `C`, we can generate a classification report for the testing set:

```
# generate a classification report for the model
print("[INFO] evaluating...")
preds = model.predict(db["features"][i:])
print(classification_report(db["labels"][i:], preds,
     target_names=db["label_names"]))

# compute the raw accuracy with extra precision
acc = accuracy_score(db["labels"][i:], preds)
print("[INFO] score: {}".format(acc))
```

And finally, the trained model can be serialized to disk for later use, if we so wish:

```
 # serialize the model to disk
print("[INFO] saving model...")
f = open(args["model"], "wb")
f.write(pickle.dumps(model.best_estimator_))
f.close()

# close the database
db.close()
```

To train our model on the ResNet50 features, simply execute the following command:

```
python train_model.py --db ../datasets/kaggle_dogs_vs_cats/hdf5/features.hdf5 \
     --model dogs_vs_cats.pickle
[INFO] tuning hyperparameters...
[INFO] best hyperparameters: {'C': 0.001}
[INFO] evaluating...
             precision    recall  f1-score   support

        cat       0.99      0.98      0.99      3160
        dog       0.98      0.99      0.99      3090

avg / total       0.99      0.99      0.99      6250

[INFO] score: 0.98688
[INFO] saving model...
```

As you can see from the output, our approach of using transfer learning via feature extraction yields an impressive accuracy of **98.69%**, enough for us to claim the #2 spot on the Kaggle Dogs vs. Cats leaderboard.

10.7 Summary

In this chapter we took a deep dive into the Kaggle Dogs vs. Cats dataset and studied to methods to obtain $> 90\%$ classification accuracy on it:

1. Training AlexNet from scratch.
2. Applying transfer learning via ResNet.

The AlexNet architecture is a seminal work first introduced by Krizhevsky et al. in 2012 [6]. Using our implementation of AlexNet, we reached 94 percent classification accuracy. This is a very respectable accuracy, especially for a network trained from scratch. Further accuracy can likely be obtained by:

1. Obtaining *more* training data.
2. Applying more aggressive data augmentation.
3. Deepening the network.

However, the 94 percent we obtained is not even enough for us to break our way into the top-25 leaderboard, let alone the top-5. Thus, to obtain our top-5 placement, we relied on transfer learning via feature extraction, specifically, the ResNet50 architecture trained on the ImageNet dataset. Since ImageNet contains *many* examples of both dog and cat breeds, applying a pre-trained network to this task is a natural, easy method to ensure we obtain higher accuracy with less effort. As our results demonstrated, we were able to obtain $\mathbf{98.69}\%$ classification accuracy, high enough to claim the *second position* on the Kaggle Dogs vs. Cats leaderboard.

The final sample chapter in this PDF is from the *ImageNet Bundle* — the *complete* deep learning for computer vision experience.

In this bundle I demonstrate how to train large-scale networks on the *massive* ImageNet dataset, apply R-CNNs and SSDs for object detection, *and much more!*

To learn how to train AlexNet on ImageNet, take a look at the sample chapter below…

5. Training AlexNet on ImageNet

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

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

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

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

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

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

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

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

5.1 Implementing AlexNet

The first step in training AlexNet on ImageNet is to implement the AlexNet architecture using the mxnet library. We have already reviewed Krizhevsky et al.'s [4] seminal architecture in Chapter 10 of the *Practitioner Bundle* where we trained AlexNet on the Kaggle Dogs vs. Cats challenge, so this network should not feel new and unfamiliar. That said, I have included Table 5.1 representing the structure of the network as a matter of completeness.

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

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

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

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

From there, we can start implementing AlexNet:

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

Layer Type	**Output Size**	**Filter Size / Stride**
INPUT IMAGE	$227 \times 227 \times 3$	
CONV	$57 \times 57 \times 96$	$11 \times 11/4 \times 4, K = 96$
ACT	$57 \times 57 \times 96$	
BN	$57 \times 57 \times 96$	
POOL	$16 \times 16 \times 96$	$3 \times 3/2 \times 2$
DROPOUT	$28 \times 28 \times 96$	
CONV	$28 \times 28 \times 256$	$5 \times 5, K = 256$
ACT	$28 \times 28 \times 256$	
BN	$28 \times 28 \times 256$	
POOL	$13 \times 13 \times 256$	$3 \times 3/2 \times 2$
DROPOUT	$13 \times 13 \times 256$	
CONV	$13 \times 13 \times 384$	$3 \times 3, K = 384$
ACT	$13 \times 13 \times 384$	
BN	$13 \times 13 \times 384$	
CONV	$13 \times 13 \times 384$	$3 \times 3, K = 384$
ACT	$13 \times 13 \times 384$	
BN	$13 \times 13 \times 384$	
CONV	$13 \times 13 \times 256$	$3 \times 3, K = 256$
ACT	$13 \times 13 \times 256$	
BN	$13 \times 13 \times 256$	
POOL	$13 \times 13 \times 256$	$3 \times 3/2 \times 2$
DROPOUT	$6 \times 6 \times 256$	
FC	4096	
ACT	4096	
BN	4096	
DROPOUT	4096	
FC	4096	
ACT	4096	
BN	4096	
DROPOUT	4096	
FC	1000	
SOFTMAX	1000	

Table 5.1: A table summary of the AlexNet architecture. Output volume sizes are included for each layer, along with convolutional filter size/pool size when relevant.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Following along with Table 5.1 above, the next step in implementing AlexNet is to define the two FC layer sets:

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

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

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

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

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

5.2 Training AlexNet

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

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

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

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

I have included an example of mxnet training log below:

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

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

5.2.1 What About Training Plots?

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

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

http://pyimg.co/ycyao

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

5.2.2 Implementing the Training Script

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Next, let's define our optimizer:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.3 Evaluating AlexNet

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

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

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

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

From there, we can parse our command line arguments:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Making predictions on the testing set is trivially easy:

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

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

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

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

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

5.4 AlexNet Experiments

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

When evaluating and comparing AlexNet performance, we normally use the BVLC AlexNet implementation provided by Caffe [12] rather than the original AlexNet implementation. This comparison is due to a number of reasons, including different data augmentations being used by Krizhevsky et al. and the usage of the (now deprecated) Local Response Normalization layers (LRNs). Furthermore, the "CaffeNet" version of AlexNet tends to be more accessible to the scientific community. In the remainder of this section I'll be comparing my results to the CaffeNet benchmark, but still referring back to the original Krizhevsky et al. paper.

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

Table 5.2: Learning rate schedule used when training AlexNet on ImageNet for Experiment #1.

5.4.1 AlexNet: Experiment #1

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

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

Notice how my batch normalization layer is now *before* the activation and I am using ReLU activation functions. I have included Table 5.2 to reflect my epoch number and associated learning rates below – we will review *why* I choose to lower the learning rates at each respective epoch in the remainder of this section.

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

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

I allowed my network to train, monitoring progress approximately every 10 epochs. One of the *worst* mistakes I see new deep learning practitioners make is checking their training plots *too often*. In most cases, you need the context of 10-15 epochs before you can make the decision that a network is indeed overfitting, underfitting, etc. After epoch 70, I plotted my training loss and accuracy (Figure 5.1), *top-left*). At this point, validation and training accuracy had essentially stagnated at $\approx 49-50\%$, a clear sign that the learning rate can be reduced to further improve accuracy.

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

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

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

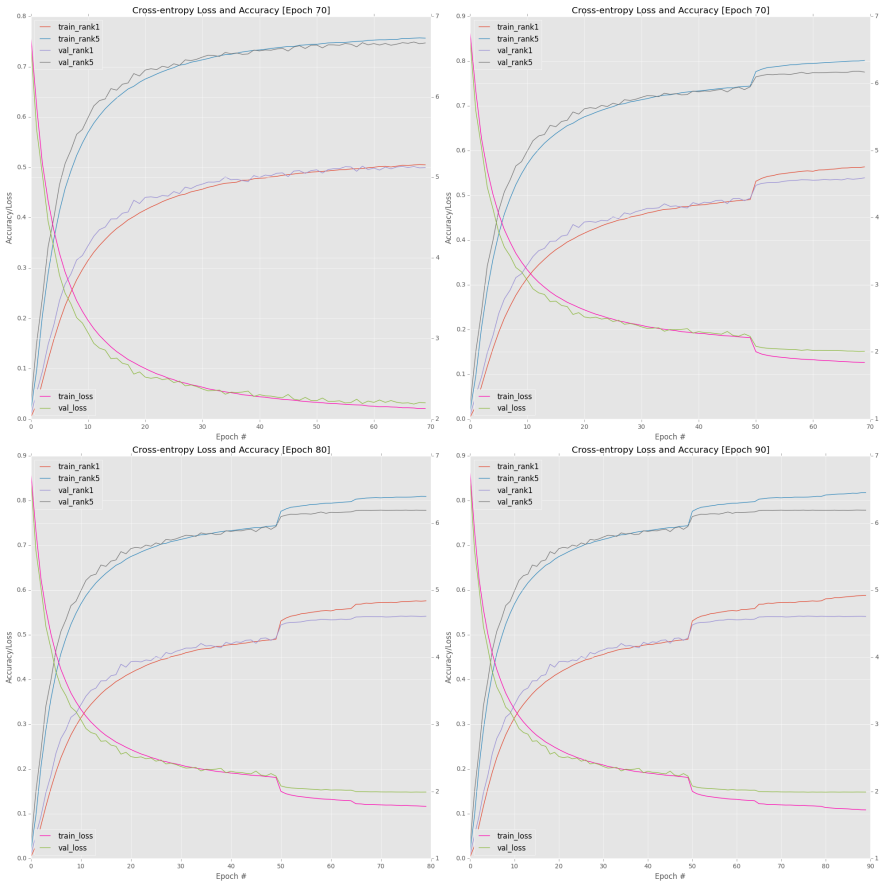

Figure 5.1: **Top-left:** Letting AlexNet train to epoch 70 with a $1e-2$ learning rate. Notice how rank-1 accuracy stagnates around $\approx 49\%$. I terminated training after epoch 70 and decided to restart training at epoch 50. **Top-right:** Restarting training from epoch 50 with a learning rate of $1e-3$. The order of magnitude decrease in α allows the network to "jump" to higher accuracy/lower loss. **Bottom-left:** Restarting training from epoch 65 with $\alpha = 1e-4$. **Bottom-right:** Epochs 80-90 at $\alpha = 1e-5$.

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

Again, I kept monitoring AlexNet progress until epoch 70 (Figure 5.1, *top-right*). The *first* key takeaway you should examine from this plot is how lowering my learning rate from $1e-2$ to $1e-3$ caused a *sharp rise* in accuracy and a *dramatic dip* in loss immediately past epoch 50 – this rise in accuracy and drop in loss is normal when you are training deep neural networks on large datasets. By lowering the learning rate we are allowing our network to descend into lower areas of loss, as previously the learning rate was too large for the optimizer to find these regions. Keep in mind that the goal of training a deep learning network is not necessarily to find a global minimum or even local minimum; rather to simply find a region where loss is sufficiently low.

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

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

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

Validation loss/accuracy improved *slightly*, but at this point, the learning rate is starting to become too small – furthermore, we are starting to overfit to the training data (Figure 5.1, *bottom-left*).

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

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

Figure 5.1 (*bottom-right*) right contains the resulting plot for the final ten epochs. Further training past epoch 90 is unnecessary as validation loss/accuracy has stopped improving while training loss continues to drop, putting us at risk at overfitting. At the end of epoch 90, I obtained 54.14% rank-1 accuracy and 77.90% rank-5 accuracy on the validation data. This accuracy is *very reasonable* for a first experiment, but not quite what I would expect from AlexNet-level performance, which the BVLC CaffeNet reference model reports to be approximately 57% rank-1 accuracy and 80% rank-5 accuracy.

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

5.4.2 AlexNet: Experiment #2

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

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

Again, I used the exact same optimizer parameters of SGD with an initial learning rate of $1e-2$, momentum of 0.9, and L2 weight decay of 0.0005. Table 5.3 includes my epoch and associated learning rate schedule. I started training AlexNet using the following command:

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

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

Table 5.3: Learning rate schedule used when training AlexNet on ImageNet for Experiment #2 and Experiment #3.

Around epoch 65 I noticed that validation loss and accuracy were stagnating (Figure 5.2, *top-left*). Therefore, I stopped training, adjusted my learning rate to be $1e-3$, and then restarted training from the 65th epoch:

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

Again, we can see the characteristic jump in accuracy by lowering the learning rate, when validation accuracy/loss plateaus (Figure 5.2, *top-right*). At epoch 85 I again lowered my learning rate, this time from $1e-3$ to $1e-4$ and allowed the network to train for 15 more epochs, after which validation loss/accuracy stopped improving (Figure 5.2, *bottom*).

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

5.4.3 AlexNet: Experiment #3

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

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

Notice how the batch normalization layer is placed *after* the activation along with ELUs replacing ReLUs. During this experiment I used the exact same SGD optimizer parameters as my previous two trials. I also followed the same learning rate schedule from the second experiment (Table 5.3).

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

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

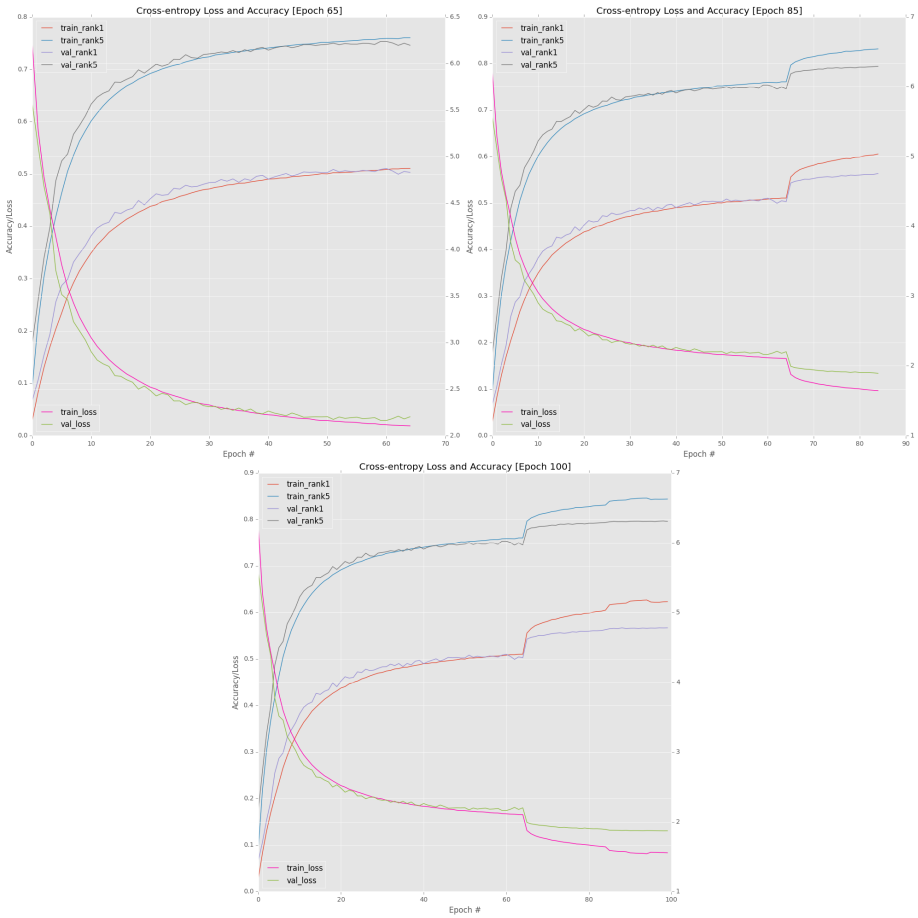

Figure 5.2: **Top-right:** The first 65 epochs with a $1e-2$ learning rate when placing the activation *before* the batch normalization. **Top-left**: Decreasing α to $1e-3$ causes a sharp increase in accuracy and decrease in loss; however, the training loss is decreasing significantly faster than validation loss. **Bottom:** A final decrease in α to $1e-4$ for epochs 85-100.

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

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

A full plot of the training and validation loss/accuracy can be seen in Figure 5.3. Again, you can see the clear characteristic marks of adjusting your learning rate by an order of magnitude at epochs 65 and 85, with the jumps becoming less pronounced as the learning rate decreases. I *did not* want to train past epoch 100 as AlexNet is clearly starting to overfit to the training data while validation accuracy/loss remains stagnate. The more this gap is allowed to grow, the worse overfitting becomes, therefore we apply the "early stopping" regularization criteria to prevent

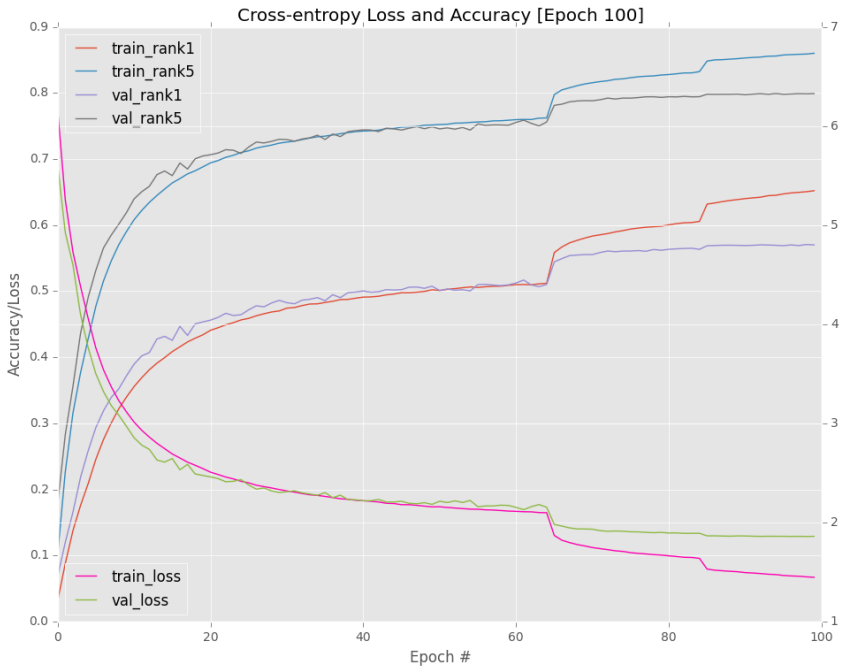

Figure 5.3: In our final AlexNet + ImageNet experiment we swap out ReLUs for ELUs and obtain a validation rank-1/rank5 accuracy of 57.00%/75.52% and testing rank-1/rank-5 accuracy of 59.80%/81.75%.

further overfitting.

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

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

I have summarized the results in Table 5.4. Here you can see that I obtained **59.80**% rank-1 and **81.75**% rank-5 accuracy on the testing set, certainly *above* what most independent papers and publications report AlexNet-level accuracy to be. For your convenience, I have included the weights for this AlexNet experiment in your download of the *ImageNet Bundle*.

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

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

Table 5.4: Evaluating AlexNet on the ImageNet test set. Our results outperform the standard Caffe reference model used to benchmark AlexNet.

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

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

5.5 Summary

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

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

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

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

It looks like you have reached the end of the table of contents + sample chapter previews!

At this point, I think you agree with me: *Deep Learning for Computer vision with Python* is the most **complete, comprehensive** guide to mastering deep learning.

You just can't find:

- Highly intuitive explanations of theory…
- Thoroughly documented code…
- Detailed experiments enabling you to reproduce state-of-the-art results…

…in any other book or online course.

Publications like this just don't exist. **Until now.**

I hope you'll join me and take a deep dive into deep learning.

With my 30-day money back guarantee you've got nothing to lose.

Click here to pick up your copy!