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

***NEW* 3rd Edition: Supports Keras and TensorFlow 2.0!**

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 950+ 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. You will learn how to train large-scale networks on *massive* datasets, train your own *custom object detectors* (Faster R-CNNs, SSDs, and RetinaNet), and *annotate + train your own Mask R-CNNs.* ***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 12-16
  - **ImageNet Bundle:** Pages 18-23

- **Sample Chapters**
  - **Pages 25-34:** What is Deep Learning?
  - **Pages 36-48:** Training Your First CNN
  - **Pages 50-69:** Case Study: Breaking captchas with a CNN
  - **Pages 71-97:** Competing in Kaggle: Dogs vs. Cats
  - **Pages 99-131:** Implementing ResNet by Hand
  - **Pages 133-158:** Mask R-CNN for Skin Lesion Segmentation

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*.

The *ImageNet Bundle* also includes *special bonus guides* on:

1. *Object detection* with Faster R-CNNs, SSDs, and RetinaNet
2. *Mask R-CNNs,* including how to prepare your image dataset and train a Mask R-CNN from scratch

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. [7]

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.

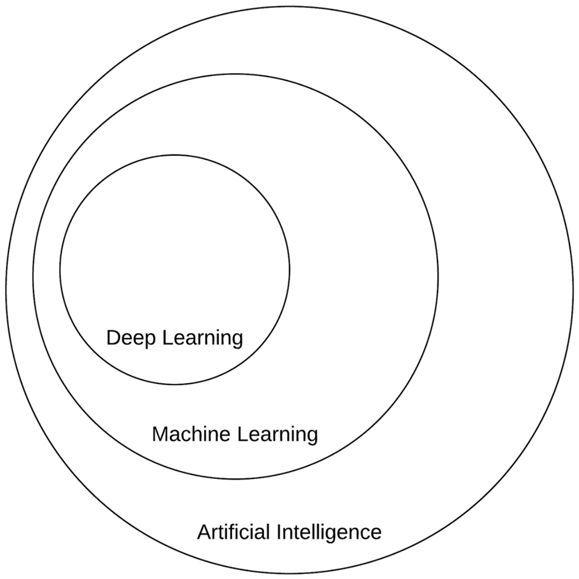


Figure 2.1: A Venn diagram describing deep learning as a subfield of machine learning which is in turn a subfield of artificial intelligence (Image inspired by Figure 1.4 of Goodfellow et al. [8]).

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 [9]. 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 [10, 11] – 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 [12] 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

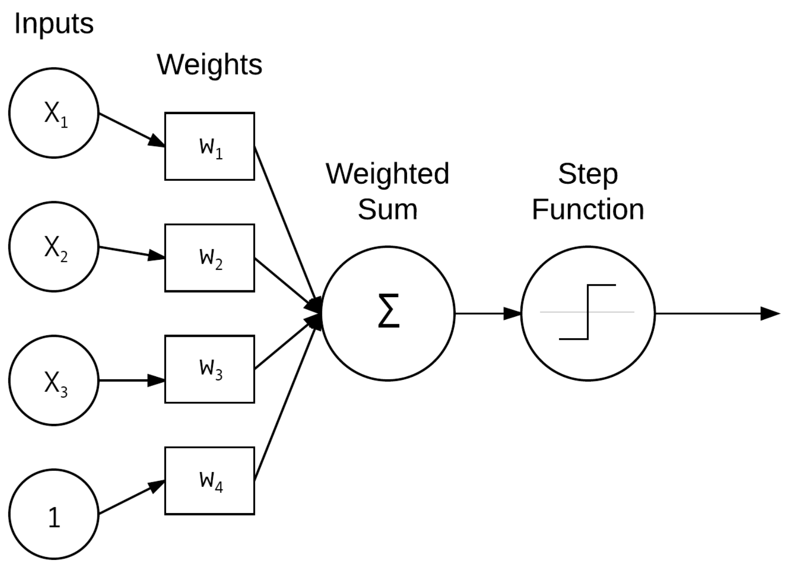


Figure 2.2: An example of the simple Perceptron network architecture that accepts a number of inputs, computes a weighted sum, and applies a step function to obtain the final prediction. We'll review the Perceptron in detail inside Chapter 10.

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) [13], Rumelhart (1986) [14], and LeCun (1998) [15] 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* [16], 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-*

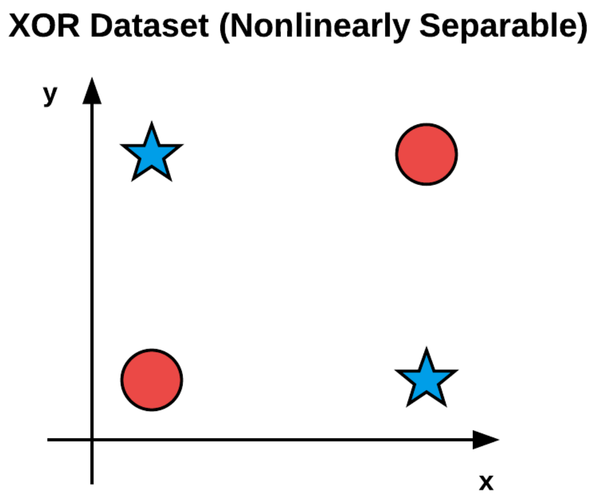


Figure 2.3: The XOR (E(X)clusive Or) dataset is an example of a nonlinear separable problem that the Perceptron *cannot* solve. Take a second to convince yourself that it is impossible to draw a single line that separates the blue stars from the red circles.

lutional Neural Network (LeCun 1988) [17] 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. [8] as well as this excellent blog post by Jason Brownlee at Machine Learning Mastery [18].

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

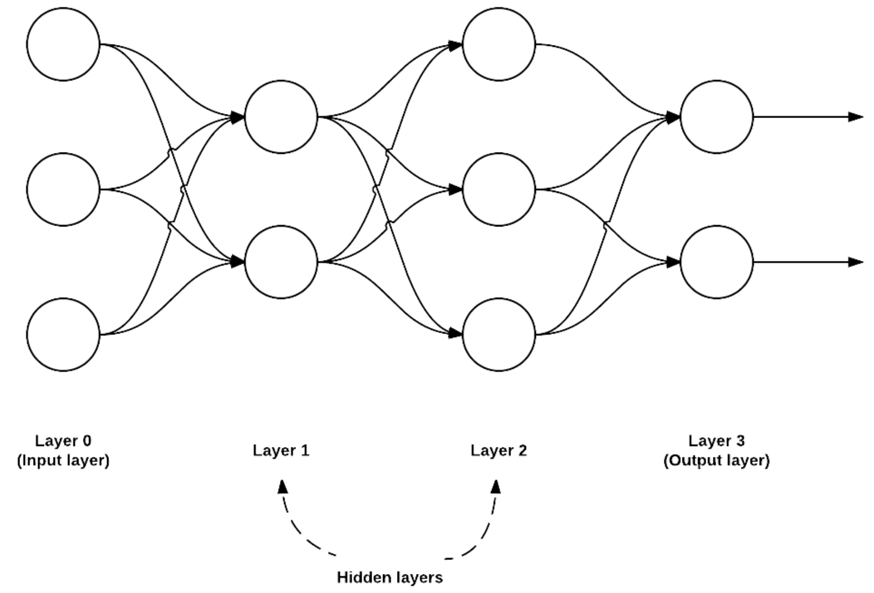


Figure 2.4: A multi-layer, feedforward network architecture with an input layer (3 nodes), two hidden layers (2 nodes in the first layer and 3 nodes in the second layer), and an output layer (2 nodes).

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 [19], Haralick texture [20]), shape (Hu Moments [21], Zernike Moments [22]), and color (color moments, color histograms, color correlograms [23]).

Other methods such as keypoint detectors (FAST [24], Harris [25], DoG [26], to name a few) and local invariant descriptors (SIFT [26], SURF [27], BRIEF [28], ORB [29], etc.) describe *salient* (i.e., the most "interesting") regions of an image.

Other methods such as Histogram of Oriented Gradients (HOG) [30] 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

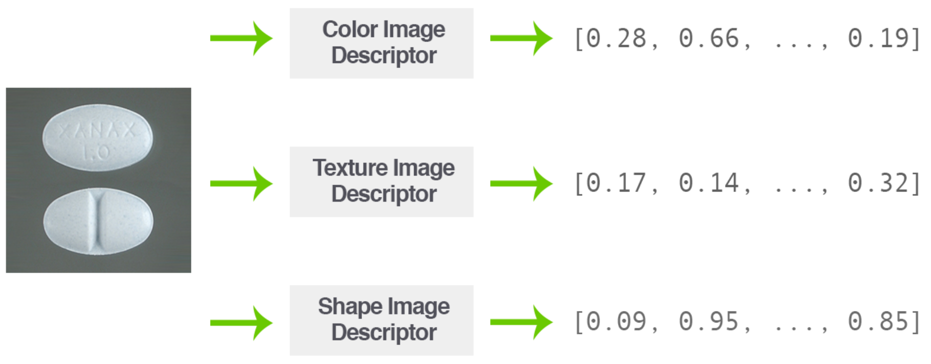


Figure 2.5: Quantifying the contents of an image containing a prescription pill medication via a series of blackbox color, texture, and shape image descriptors.

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 [32] and Exemplar SVMs [33].

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 [31].

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

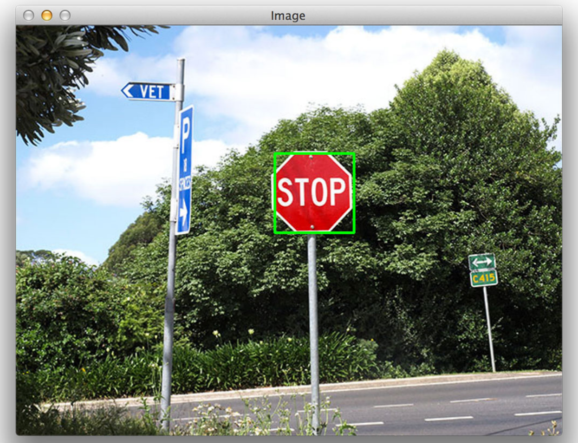


Figure 2.6: The HOG + Linear SVM object detection framework applied to detecting the location of stop signs in images, as covered inside the PyImageSearch Gurus course [31].

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* [34]:

> *"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 [8].

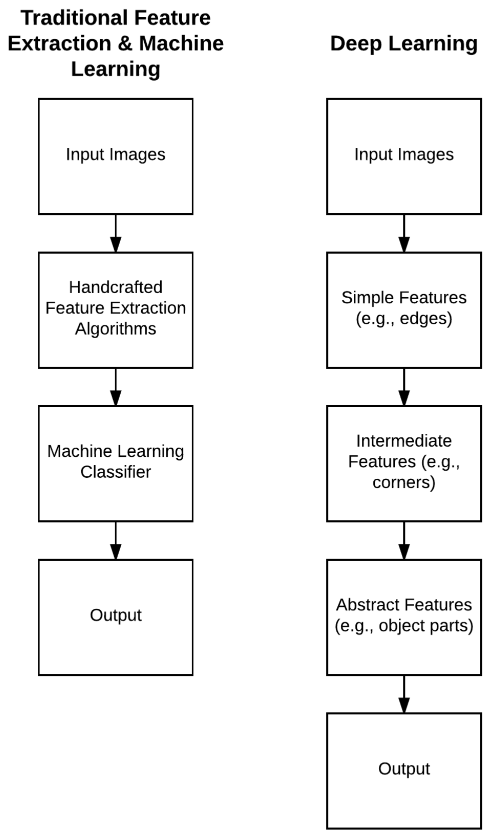

Traditional Feature Extraction & Machine Learning
Deep Learning
Input Images
Handcrafted Feature Extraction Algorithms
Machine Learning Classifier
Output
Input Images
Simple Features (e.g., edges)
Intermediate Features (e.g., corners)
Abstract Features (e.g., object parts)
Output

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* [35], 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* [36], 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*, [37] can be seen in Figure 2.8, providing an example of this behavior.

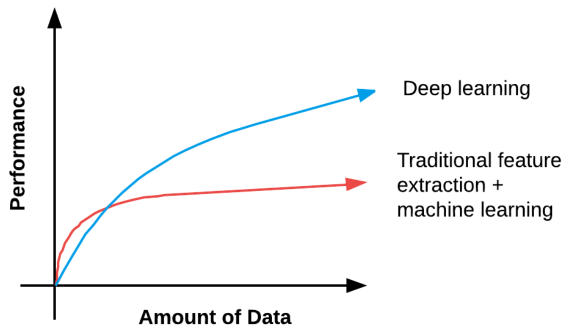


Figure 2.8: As the amount of data available to deep learning algorithms increases, accuracy does as well, substantially outperforming traditional feature extraction + machine learning approaches.

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** [38].

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 with 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 tensorflow.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 be 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 `SimplePreprocessor` from Chapter 7:

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

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

```
2  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:

```
3  sdl = SimpleDatasetLoader(preprocessors=[sp, iap])
4  (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

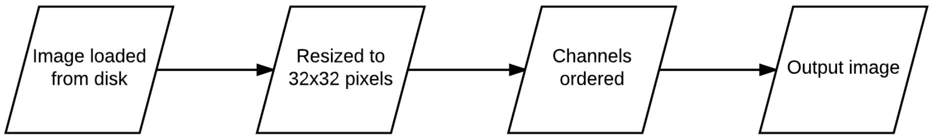


Figure 12.1: An example image pre-processing pipeline that (1) loads an image from disk, (2) resizes it to 32×32 pixels, (3) orders the channel dimensions, and (4) outputs the image.

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:

```python
# import the necessary packages
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D
```

```
4  from tensorflow.keras.layers import Activation
5  from tensorflow.keras.layers import Flatten
6  from tensorflow.keras.layers import Dense
7  from tensorflow.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 22** 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:

```
        # softmax classifier
        model.add(Flatten())
        model.add(Dense(classes))
        model.add(Activation("softmax"))

        # return the constructed network architecture
        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:

```
# import the necessary packages
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from pyimagesearch.preprocessing import ImageToArrayPreprocessor
from pyimagesearch.preprocessing import SimplePreprocessor
from pyimagesearch.datasets import SimpleDatasetLoader
from pyimagesearch.nn.conv import ShallowNet
from tensorflow.keras.optimizers import SGD
from imutils import paths
import matplotlib.pyplot as plt
import numpy as np
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 `SimplePreprocessor` 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 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["accuracy"], label="train_acc")
69 plt.plot(np.arange(0, 100), H.history["val_accuracy"], 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.

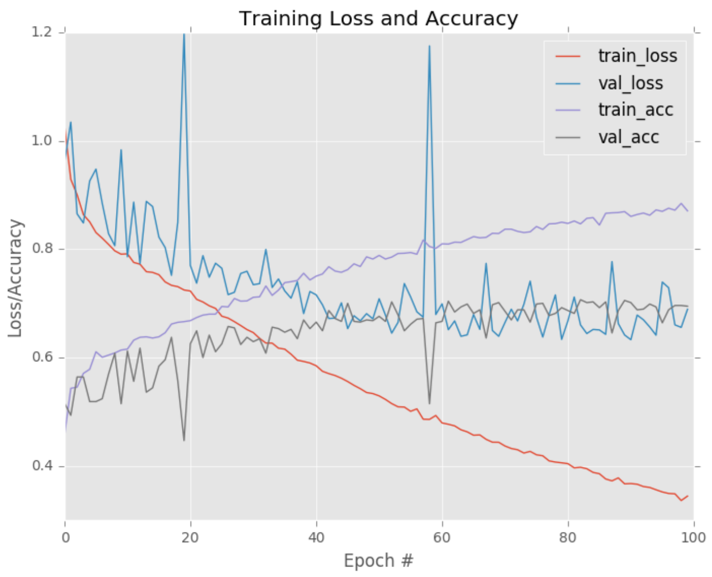


Figure 12.2: A plot of our loss and accuracy over the course of 100 epochs for the ShallowNet architecture trained on the Animals dataset.

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:

```python
# import the necessary packages
from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import classification_report
from pyimagesearch.nn.conv import ShallowNet
from tensorflow.keras.optimizers import SGD
from tensorflow.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")
```

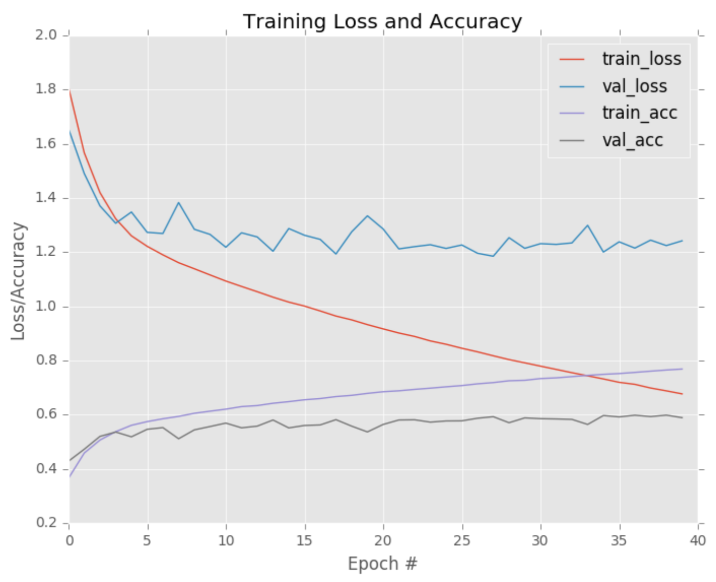


Figure 12.3: Loss and accuracy for ShallowNet trained on CIFAR-10. Our network obtains 60% classification accuracy; however, it is overfitting. Further accuracy can be obtained by applying regularization, which we'll cover later in this book.

```
plt.plot(np.arange(0, 40), H.history["accuracy"], label="train_acc")
plt.plot(np.arange(0, 40), H.history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
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, an increase of the previous best of 57% using simple multi-layer NNs (and without the *significant* overfitting).

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

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.

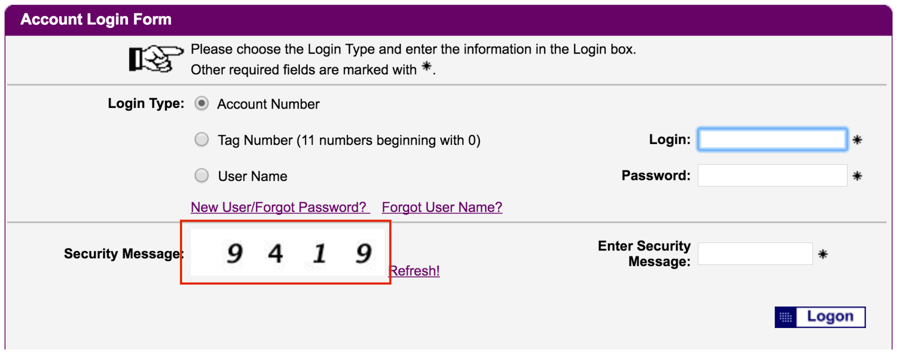
21.1.1 A Note on Responsible Disclosure

Living in the northeastern/midwestern part of the United States, it's hard to travel on major highways without an E-ZPass [137]. 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


Account Login Form

Please choose the Login Type and enter the information in the Login box.
Other required fields are marked with *.

Login Type: [✓ Account Number] Tag Number (11 numbers beginning with 0) User Name

Login: *
Password: *

New User/Forgot Password? Forgot User Name?

Security Message: 9 4 1 9 Refresh!

Enter Security Message: *

Logon

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, these 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*).

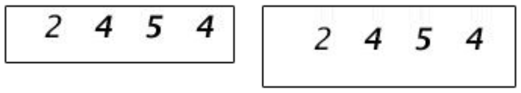


Figure 21.2: **Left:** The original image loaded from disk. **Right:** Padding the image to ensure we can extract the digits *just in case* any of the digits are touching the border of the image.

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.

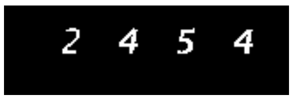


Figure 21.3: Thresholding the image ensures the foreground is *white* while the background is *black*. This is a typical assumption/requirement when working with many image processing functions with OpenCV.

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 large 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.

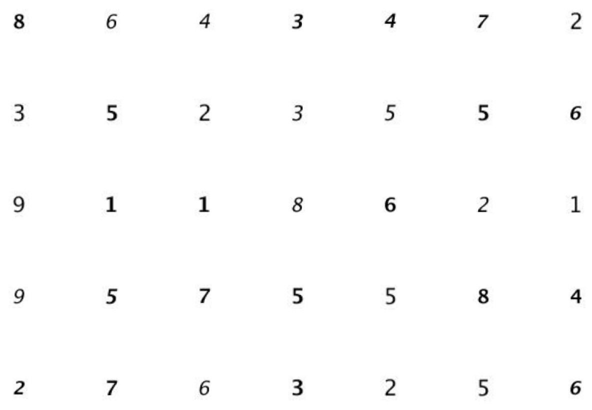


Figure 21.4: A sample of the digit ROIs extracted from our captcha images. Our goal will be to label these images in such a way that we can train a custom Convolutional Neural Network on them.

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.

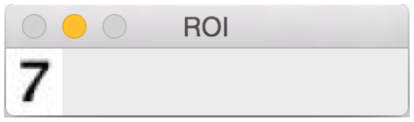


Figure 21.5: When annotating our dataset of digits, a given digit ROI will display on our screen. We then need to press the corresponding key on our keyboard to label the image and save the ROI to disk.

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 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 tensorflow.keras.preprocessing.image import img_to_array
6  from tensorflow.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)
```

```

        # extract the class label from the image path and update the
        # labels list
        label = imagePath.split(os.path.sep)[-2]
        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:

```
# scale the raw pixel intensities to the range [0, 1]
data = np.array(data, dtype="float") / 255.0
labels = np.array(labels)

# partition the data into training and testing splits using 75% of
# the data for training and the remaining 25% for testing
(trainX, testX, trainY, testY) = train_test_split(data,
    labels, test_size=0.25, random_state=42)

# convert the labels from integers to vectors
lb = LabelBinarizer().fit(trainY)
trainY = lb.transform(trainY)
testY = lb.transform(testY)
```

We can then initialize the LeNet model and SGD optimizer:

```
# initialize the model
print("[INFO] compiling model...")
model = LeNet.build(width=28, height=28, depth=1, classes=9)
opt = SGD(lr=0.01)
model.compile(loss="categorical_crossentropy", optimizer=opt,
    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["accuracy"], label="acc")
plt.plot(np.arange(0, 15), H.history["val_accuracy"], 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.

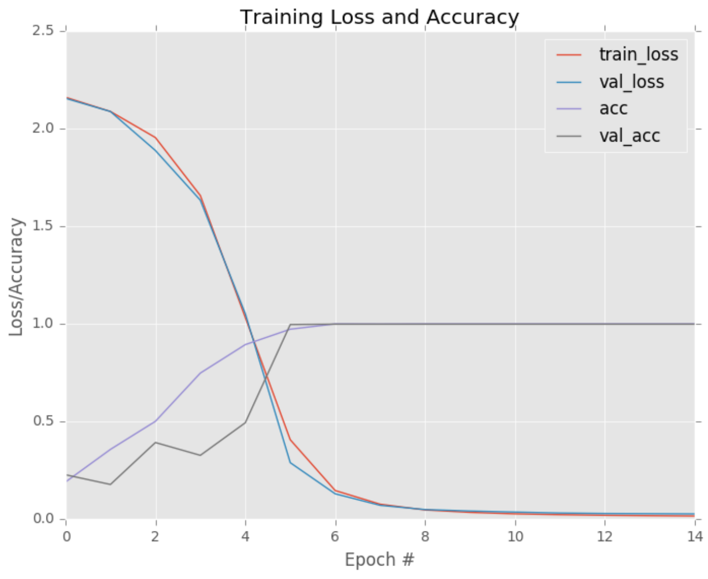


Figure 21.6: Using the LeNet architecture on our custom digits datasets enables us to obtain 100% classification accuracy after only fifteen epochs. Furthermore, there are no signs of overfitting.

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:

```python
# import the necessary packages
from tensorflow.keras.preprocessing.image import img_to_array
from tensorflow.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:

```python
# 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:

```python
# 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:

```python
# 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 near 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 then classify it
```

```
64         roi = preprocess(roi, 28, 28)
65         roi = np.expand_dims(img_to_array(roi), axis=0) / 255.0
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 custom Convolutional Neural Network on our labeled dataset.
4. Test and evaluate our model on example images.

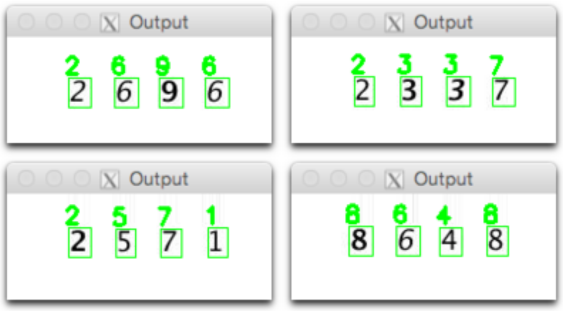


Figure 21.7: Examples of captchas that have been correctly classified and broken by our LeNet model.

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 cross-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* [6]. 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 three 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 *training* 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.

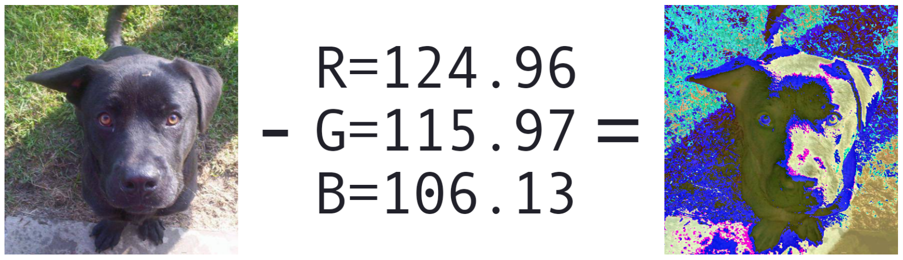


Figure 10.1: An example of applying mean subtraction to an input image (*left*) by subtracting $R = 124.96$, $G = 115.97$, $B = 106.13$ pixel-wise, resulting in the output image (*right*). Mean subtraction is used to reduce the affects of lighting variations during classification.

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).

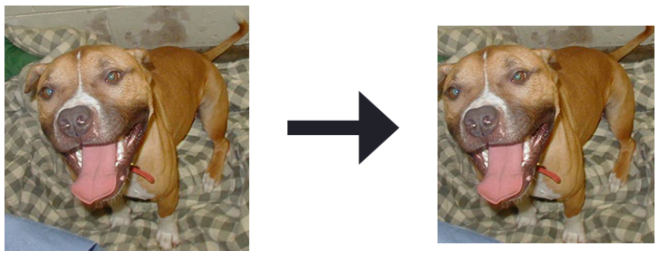


Figure 10.2: **Left:** Our original 256×256 input image. **Right:** *Randomly* cropping a 227×227 region from the image.

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:

```python
# import the necessary packages
from sklearn.feature_extraction.image import extract_patches_2d

class PatchPreprocessor:
     def __init__(self, width, height):
          # store the target width and height of the image
          self.width = width
          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:

```
    def preprocess(self, image):
        # extract a random crop from the image with the target width
        # and height
        return extract_patches_2d(image, (self.height, self.width),
            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).

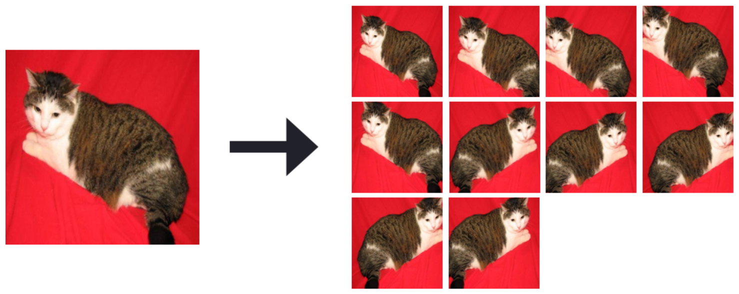


Figure 10.3: **Left:** The original 256×256 input image. **Right:** Applying the 10-crop preprocessor to extract ten 227×227 crops of the image including the center, four corners, and their corresponding horizontal mirrors.

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 tensorflow.keras.utils import to_categorical
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, "r")
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 = 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 11×11 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	$55 \times 55 \times 96$	$11 \times 11/4 \times 4, K = 96$
ACT	$55 \times 55 \times 96$	
BN	$55 \times 55 \times 96$	
POOL	$27 \times 27 \times 96$	$3 \times 3/2 \times 2$
DROPOUT	$27 \times 27 \times 96$	
CONV	$27 \times 27 \times 256$	$5 \times 5, K = 256$
ACT	$27 \times 27 \times 256$	
BN	$27 \times 27 \times 256$	
POOL	$13 \times 13 \times 256$	$3 \times 3/2 \times 2$
DROPOUT	$13 \times 13 \times 256$	
CONV	$13 \times 13 \times 384$	$3 \times 3, K = 384$
ACT	$13 \times 13 \times 384$	
BN	$13 \times 13 \times 384$	
CONV	$13 \times 13 \times 384$	$3 \times 3, K = 384$
ACT	$13 \times 13 \times 384$	
BN	$13 \times 13 \times 384$	
CONV	$13 \times 13 \times 256$	$3 \times 3, K = 256$
ACT	$13 \times 13 \times 256$	
BN	$13 \times 13 \times 256$	
POOL	$13 \times 13 \times 256$	$3 \times 3/2 \times 2$
DROPOUT	$6 \times 6 \times 256$	
FC	4096	
ACT	4096	
BN	4096	
DROPOUT	4096	
FC	4096	
ACT	4096	
BN	4096	
DROPOUT	4096	
FC	1000	
SOFTMAX	1000	

Table 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 tensorflow.keras.models import Sequential
3  from tensorflow.keras.layers import BatchNormalization
4  from tensorflow.keras.layers import Conv2D
5  from tensorflow.keras.layers import MaxPooling2D
6  from tensorflow.keras.layers import Activation
7  from tensorflow.keras.layers import Flatten
8  from tensorflow.keras.layers import Dropout
9  from tensorflow.keras.layers import Dense
10 from tensorflow.keras.regularizers import l2
11 from tensorflow.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:

```
        # Block #3: CONV => RELU => CONV => RELU => CONV => RELU
        model.add(Conv2D(384, (3, 3), padding="same",
            kernel_regularizer=l2(reg)))
        model.add(Activation("relu"))
        model.add(BatchNormalization(axis=chanDim))
        model.add(Conv2D(384, (3, 3), padding="same",
            kernel_regularizer=l2(reg)))
        model.add(Activation("relu"))
        model.add(BatchNormalization(axis=chanDim))
        model.add(Conv2D(256, (3, 3), padding="same",
            kernel_regularizer=l2(reg)))
        model.add(Activation("relu"))
        model.add(BatchNormalization(axis=chanDim))
        model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))
        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 top 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):

```
        # Block #4: first set of FC => RELU layers
        model.add(Flatten())
        model.add(Dense(4096, kernel_regularizer=l2(reg)))
        model.add(Activation("relu"))
        model.add(BatchNormalization())
        model.add(Dropout(0.5))

        # Block #5: second set of FC => RELU layers
        model.add(Dense(4096, kernel_regularizer=l2(reg)))
        model.add(Activation("relu"))
        model.add(BatchNormalization())
        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:

```
         # softmax classifier
        model.add(Dense(classes, kernel_regularizer=l2(reg)))
        model.add(Activation("softmax"))

        # return the constructed network architecture
        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 tensorflow.keras.preprocessing.image import ImageDataGenerator
16 from tensorflow.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 implemented on **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.

R Make sure you copy your `dogs_vs_cats_mean.json` file over from the previous chapter!

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=10,
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 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.

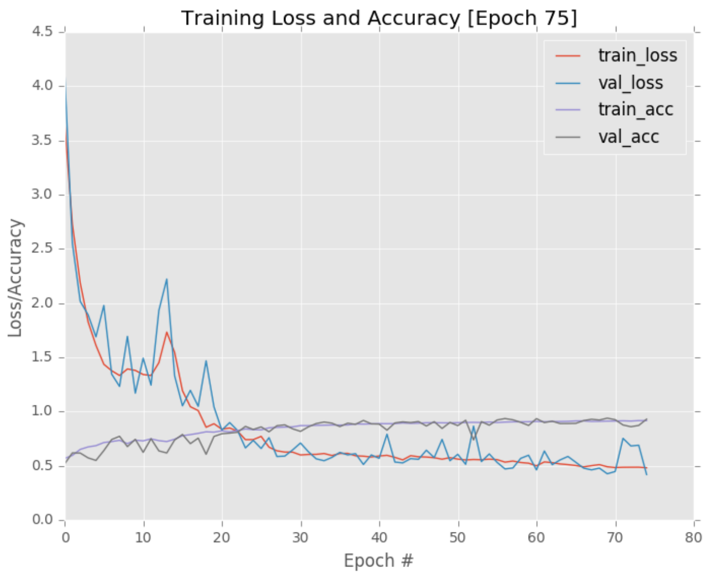


Figure 10.4: Training AlexNet on the Kaggle Dogs vs. Cats competition where we obtain 92.97% classification accuracy on our validation set. Our learning curve is stable with changes in training accuracy/loss being reflected in the respective validation split.

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 tensorflow.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=10)
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:

- Center of the image
- 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:

```
# compute the rank-1 accuracy
pbar.finish()
print("[INFO] predicting on test data (with crops)...")
(rank1, _) = rank5_accuracy(predictions, testGen.db["labels"])
print("[INFO] rank-1: {:.2f}%".format(rank1 * 100))
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 the #1 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 tensorflow.keras.applications import ResNet50
3  from tensorflow.keras.applications import imagenet_utils
4  from tensorflow.keras.preprocessing.image import img_to_array
5  from tensorflow.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
```

```
print("[INFO] loading images...")
imagePaths = list(paths.list_images(args["dataset"]))
random.shuffle(imagePaths)

# extract the class labels from the image paths then encode the
# labels
labels = [p.split(os.path.sep)[-1].split(".")[0] for p in imagePaths]
le = LabelEncoder()
labels = le.fit_transform(labels)
```

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

```
# load the ResNet50 network
print("[INFO] loading network...")
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:

```
# initialize the HDF5 dataset writer, then store the class label
# names in the dataset
dataset = HDF5DatasetWriter((len(imagePaths), 100352),
     args["output"], dataKey="features", bufSize=args["buffer_size"])
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:

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

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

Followed by pre-processing each image:

```python
        # loop over the images and labels in the current batch
        for (j, imagePath) in enumerate(batchPaths):
            # load the input image using the Keras helper utility
            # while ensuring the image is resized to 224x224 pixels
            image = load_img(imagePath, target_size=(224, 224))
            image = img_to_array(image)

            # preprocess the image by (1) expanding the dimensions and
            # (2) subtracting the mean RGB pixel intensity from the
            # ImageNet dataset
            image = np.expand_dims(image, axis=0)
            image = imagenet_utils.preprocess_input(image)

            # add the image to the batch
            batchImages.append(image)
```

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

```python
        # pass the images through the network and use the outputs as
        # our actual features
        batchImages = np.vstack(batchImages)
        features = model.predict(batchImages, batch_size=bs)

        # reshape the features so that each image is represented by
        # a flattened feature vector of the `MaxPooling2D` outputs
        features = features.reshape((features.shape[0], 100352))
```

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

```python
        # add the features and labels to our HDF5 dataset
        dataset.add(features, batchLabels)
        pbar.update(i)

# close the dataset
dataset.close()
pbar.finish()
```

R In Keras v2.2.0 and greater the output dimensions of ResNet is $2,048 \times 7 \times 7 = 100,352$ rather than previous versions which were $2,048$. If you are using a version of Keras prior to v2.2.0+ make sure you update the 100352 in both the `.reshape` and `HDF5DatasetWriter` calls above to be 2048.

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 `features.hdf5` in your `hdf5` directory:

```
$ ls -l ../datasets/kaggle_dogs_vs_cats/hdf5/features.hdf5
-rw-rw-r-- adrian 409806272 Jun  3 07:17 features.hdf5
```

Given these features, we can train a Logistic Regression classifier 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:

```
# define the set of parameters that we want to tune then start a
# grid search where we evaluate our model for each value of C
print("[INFO] tuning hyperparameters...")
params = {"C": [0.0001, 0.001, 0.01, 0.1, 1.0]}
model = GridSearchCV(LogisticRegression(solver="lbfgs",
     multi_class="auto"), params, cv=3, n_jobs=args["jobs"])
model.fit(db["features"][:i], db["labels"][:i])
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': 0.0001}
[INFO] evaluating...
              precision    recall  f1-score   support

         cat       0.99      0.99      0.99      3126
         dog       0.99      0.99      0.99      3124

    accuracy                           0.99      6250
   macro avg       0.99      0.99      0.99      6250
weighted avg       0.99      0.99      0.99      6250

[INFO] score: 0.99152
[INFO] saving model..
```

As you can see from the output, our approach of using transfer learning via feature extraction yields an impressive accuracy of **99.15%**, enough for us to claim the #1 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 two 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 **99.15%** classification accuracy, high enough to claim *first place* on the Kaggle Dogs vs. Cats leaderboard.

Learning how to implement state-of-the-art architectures is an important skill to learn, enabling you to create *your own models* and *successfully train them.*

This chapter from the *Practitioner Bundle* shows you how to implement ResNet from scratch…

12. ResNet

In our previous chapter, we discussed the GoogLeNet architecture and the Inception module, a *micro-architecture* that acts as a building block in the overall *macro-architecture*. We are now going to discuss another network architecture that relies on micro-architectures – *ResNet*.

ResNet uses what's called a *residual module* to train Convolutional Neural Networks to depths previously thought impossible. For example, in 2014, the VGG16 and VGG19 architectures were considered *very* deep [11]. However, with ResNet, we have successfully trained networks with >100 *layers* on the challenging ImageNet dataset and *over 1,000 layers* on CIFAR-10 [24].

These depths are only made possible by using "smarter" weight initialization algorithms (such as Xavier/Glorot [44] and MSRA/He et al. [45]) along with *identity mapping*, a concept we'll discuss later in this chapter. Given the depths of ResNet networks, perhaps it comes as no surprise that ResNet took first place in *all three* ILSVRC 2015 challenges (classification, detection, and localization).

In this chapter, we are going to discuss the ResNet architecture, the residual module, along with updates to the residual module that have made it capable of obtaining higher classification accuracy. From there we'll implement and train variants of ResNet on the CIFAR-10 dataset and the Tiny ImageNet challenge – in each case, our ResNet implementations will outperform every experiment we have executed in this book.

12.1 ResNet and the Residual Module

First introduced by He et al. in their 2015 paper, *Deep Residual Learning for Image Recognition* [24], the ResNet architecture has become a seminal work, demonstrating that *extremely deep* networks can be trained using standard SGD and a reasonable initialization function. In order to train networks at depths greater than 50-100 (and in some cases, 1,000) layers, ResNet relies on a micro-architecture called the *residual module*.

Another interesting component of ResNet is that pooling layers are used *extremely sparingly*. Building on the work of Springenberg et al. [41], ResNet *does not* strictly rely on max pooling operations to reduce volume size. Instead, convolutions with strides >1 are used to not only learn weights, but reduce the output volume spatial dimensions. In fact, there are only two occurrences of pooling being applied in the full implementation of the architecture:

1. The first (and only) occurrence of max pooling happens early in the network to help reduce spatial dimensions.
2. The second pooling operation is actually an *average pooling layer* used in place of fully-connected layers, like in GoogLeNet.

Strictly speaking, there is only *one* max pooling layer – all other reductions in spatial dimensions are handled by convolutional layers.

In this section, we'll review the *original* residual module, along with the bottleneck residual module used to train deeper networks. From there, we'll discuss extensions and updates to the original residual module by He et al. in their 2016 publication, *Identity Mappings in Deep Residual Networks* [33], that allow us to further increase classification accuracy. Later in this chapter, we'll implement ResNet from scratch using Keras.

12.1.1 Going Deeper: Residual Modules and Bottlenecks

The original residual module introduced by He et al. in 2015 relies on *identity mappings*, the process of taking the *original input* to the module and adding it to the *output* of a series of operations. A graphical depiction of this module can be seen in Figure 12.1 (*left*). Notice how this module only has two branches, unlike the four branches in the Inception module of GoogLeNet. Furthermore, this module is *highly simplistic*.

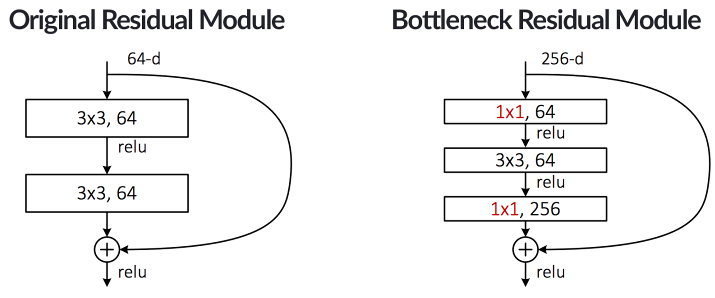


Figure 12.1: **Left:** The original Residual module proposed by He et al. **Right:** The more commonly used bottleneck variant of the Residual module.

At the top of the module, we accept an *input* to the module (i.e., the previous layer in the network). The right branch is a "linear shortcut" – it connects the input to an addition operation at the bottom of the module. Then, on the left branch of the residual module, we apply a series of convolutions (all of which are 3×3), activations, and batch normalizations. This is a fairly standard pattern to follow when constructing Convolutional Neural Networks.

But what makes ResNet interesting is that He et al. suggested *adding* the *original input* to the *output* of the `CONV`, `RELU` and `BN` layers. We call this addition an *identity mapping* since the input (the identity) is added to the output of series of operations. It is also why the term "residual" is used. The "residual" input is added to the output of a series of layer operations. The connection between the input and the addition node is called the ***shortcut***. Note that we are *not* referring to concatenation along the channel dimension as we have done in previous chapters. Instead, we are performing simple $1+1=2$ addition at the bottom of the module between the two branches.

While traditional neural network layers can be seen as learning a function $y = f(x)$, a residual layer attempts to approximate y via $f(x) + id(x) = f(x) + x$ where $id(x)$ is the identity function.

These residual layers *start* at the identity function and *evolve* to become more complex as the network learns. This type of residual learning framework allows us to train networks that are *substantially deeper* than previously proposed network architectures.

Furthermore, since the input is included in every residual module, it turns out the network can learn *faster* and with *larger learning rates*. It is very common to see the base learning rates for ResNet implementations start at $1e-1$. For most architectures such as AlexNet or VGGNet, this high of a learning rate would almost guarantee the network would not converge. But since ResNet relies on residual modules via identity mappings, this higher learning rate is completely possible.

In the same 2015 work, He et al. also included an extension to the original residual module called ***bottlenecks*** (Figure 12.1, *right*). Here we can see that the same identity mapping is taking place, only now the CONV layers in the left branch of the residual module have been updated:

1. We are utilizing three CONV layers rather than just two.
2. The first and last CONV layers are 1×1 convolutions.
3. The number of filters learned in the first two CONV layers are $1/4$ the number of filters learned in the final CONV.

To understand why we call this a "bottleneck", consider the following figure where two residual modules are stacked on top of each other, with one residual feeding into the next (Figure 12.2).

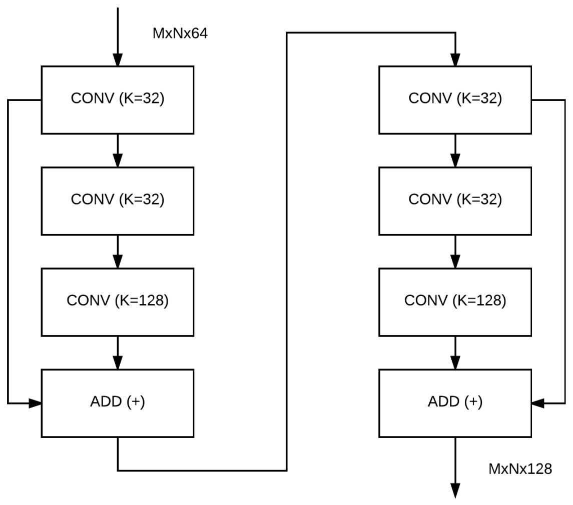


Figure 12.2: An example of two stacked residual modules where one feeds into next. Both modules learn $K = 32$, 32, and 128 filters, respectively. Notice how the dimensionality is reduced during the first two CONV layers then increased during the final CONV layer.

The first residual module accepts an input volume of size $M \times N \times 64$ (the actual width and height are arbitrary for this example). The three CONV layers in the first residual module learn $K = 32$, 32, and 128 filters, respectively. After applying the first residual module our output volume size is $M \times N \times 128$ which is then fed into the second residual module.

In the second residual module, our number of filters learned by each of the three CONV layers stays the same at $K = 32$, 32, and 128, respectively. However, notice that $32 < 128$, implying that we are actually reducing the volume size during the 1×1 and 3×3 CONV layers. This result has the benefit of leaving the 3×3 bottleneck layer with smaller input and output dimensions.

The final 1×1 CONV then applies $4x$ the number of filters than the first two CONV layers, thereby increasing dimensionality once again, which is why we call this update to the residual module the “bottleneck” technique. When building our own residual modules, it’s common to supply pseudocode such as `residual_module(K=128)` which implies that the *final* `CONV` layer will learn 128 filters, while the first two will learn $128/4 = 32$ filters. This notation is often easier to work with as it’s understood that the bottleneck `CONV` layers will learn $1/4th$ the number of filters as the final `CONV` layer.

When it comes to training ResNet, we typically use the bottleneck variant of the residual module rather than the original version, *especially* for ResNet implementations with > 50 layers.

12.1.2 Rethinking the Residual Module

In 2016, He et al. published a second paper on the residual module entitled *Identity Mappings in Deep Residual Networks* [33]. This publication described a comprehensive study, both theoretically and empirically, on the ordering of convolutional, activation, and batch normalization layers *within* the residual module itself. Originally, the residual module (with bottleneck) looked like Figure 12.3 (*left*).

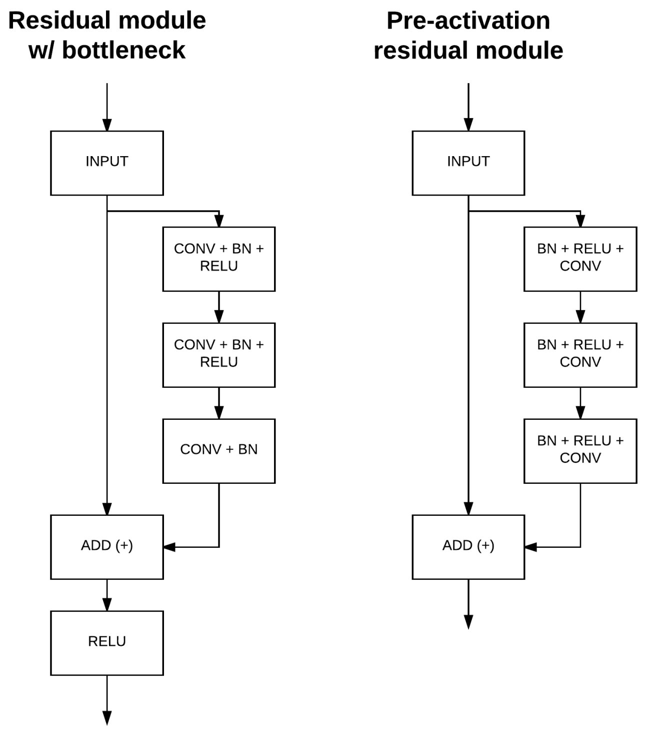


Figure 12.3: **Left:** The original residual module with bottleneck. **Right:** Adapting the bottleneck module to use pre-activations.

The original residual module with bottleneck accepts an input (a ReLU activation map) and then applies a series of `(CONV => BN => RELU) * 2 => CONV => BN` before adding this output to the original input and applying a final ReLU activation (which is then fed into the next residual module in the network). However, the He et al. 2016 study, it was found there was a more optimal layer ordering capable of obtaining higher accuracy – this method is called ***pre-activation***.

In the pre-activation version of the residual module, we remove the ReLU at the bottom of the module and re-order the batch normalization and activation such that they come *before* the convolution (Figure 12.3, *right*).

Now, instead of starting with a convolution, we apply a series of `(BN => RELU => CONV) * 3` (assuming the bottleneck is being used, of course). The output of the residual module is now the *addition operation* which is subsequently fed into the next residual module in the network (since residual modules are stacked on top of each other).

We call this layer ordering *pre-activation* as our ReLUs and batch normalization are placed *before* the convolutions, which is in contrast to the typical approach of applying ReLUs and batch normalizations *after* the convolutions. In our next section, we'll implement ResNet from scratch using both bottlenecks and pre-activations.

12.2 Implementing ResNet

Now that we have reviewed the ResNet architecture, let's go ahead and implement in Keras. For this specific implementation, we'll be using the most recent incarnation of the residual module, including bottlenecks and pre-activations. To update your project structure, create a new file named `resnet.py` inside the `nn.conv` sub-module of `pyimagesearch` – that is where our `ResNet` implementation will live:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |--- io
|    |--- nn
|    |    |--- __init__.py
|    |    |--- conv
|    |    |    |--- __init__.py
|    |    |    |--- alexnet.py
|    |    |    |--- deepergooglenet.py
|    |    |    |--- lenet.py
|    |    |    |--- minigooglenet.py
|    |    |    |--- minivggnet.py
|    |    |    |--- fcheadnet.py
|    |    |    |--- resnet.py
|    |    |    |--- shallownet.py
|    |--- preprocessing
|    |--- utils
```

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

```python
# import the necessary packages
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import AveragePooling2D
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.layers import ZeroPadding2D
from tensorflow.keras.layers import Activation
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Input
from tensorflow.keras.models import Model
```

```
12 from tensorflow.keras.layers import add
13 from tensorflow.keras.regularizers import l2
14 from tensorflow.keras import backend as K
```

We start off by importing our fairly standard set of classes and functions when building Convolutional Neural Networks. However, I would like to draw your attention to **Line 12** where we import the `add` function. Inside the residual module, we'll need to add together the outputs of two branches, which will be accomplished via this `add` method. We'll also import the `l2` function on **Line 13** so that we can perform L2 weight decay. Regularization is *extremely* important when training ResNet since, due to the network's depth, it is prone to overfitting.

Next, let's move on to our `residual_module`:

```
16 class ResNet:
17     @staticmethod
18     def residual_module(data, K, stride, chanDim, red=False,
19         reg=0.0001, bnEps=2e-5, bnMom=0.9):
```

This specific implementation of ResNet was inspired by both He et al. in their Caffe distribution [46] as well as the mxnet implementation from Wei Wu [47], therefore we will follow their parameter choices as closely as possible. Looking at the `residual_module` we can see that the function accepts more parameters than any of our previous functions – let's review each of them in detail.

The `data` parameter is simply the *input* to the residual module. The value `K` defines the number of filters that will be learned by the *final* `CONV` in the bottleneck. The first two `CONV` layers will learn `K / 4` filters, as per the He et al. paper. The `stride` controls the stride of the convolution. We'll use this parameter to help us reduce the spatial dimensions of our volume *without* resorting to max pooling.

We then have the `chanDim` parameter which defines the axis which will perform batch normalization – this value is specified later in the `build` function based on whether we are using "channels last" or "channels first" ordering.

Not all residual modules will be responsible for reducing the dimensions of our spatial volume – the `red` (i.e., "reduce") boolean will control whether we are reducing spatial dimensions (`True`) or not (`False`).

We can then supply a regularization strength to all `CONV` layers in the residual module via `reg`. The `bnEps` parameter controls the ε responsible for avoiding "division by zero" errors when normalizing inputs. In Keras, ε defaults to 0.001; however, for our particular implementation, we'll allow this value to be reduced significantly. The `bnMom` controls the momentum for the moving average. This value normally defaults to 0.99 inside Keras, but He et al. as well as Wei Wu recommend decreasing the value to 0.9.

Now that the parameters of `residual_module` are defined, let's move on to the body of the function:

```
20         # the shortcut branch of the ResNet module should be
21         # initialize as the input (identity) data
22         shortcut = data
23 
24         # the first block of the ResNet module are the 1x1 CONVs
25         bn1 = BatchNormalization(axis=chanDim, epsilon=bnEps,
26             momentum=bnMom)(data)
27         act1 = Activation("relu")(bn1)
```

```
28          conv1 = Conv2D(int(K * 0.25), (1, 1), use_bias=False,
29              kernel_regularizer=l2(reg))(act1)
```

On **Line 22** we initialize the `shortcut` in the residual module, which is simply a reference to the input `data`. We will later add the `shortcut` to the output of our bottleneck + pre-activation branch.

The first pre-activation of the bottleneck branch can be seen in **Lines 25-29**. Here we apply a batch normalization layer, followed by ReLU activation, and then a 1×1 convolution, using $K/4$ total filters. You'll also notice that we are *excluding* the bias term from our `CONV` layers via `use_bias=False`. Why might we wish to purposely leave out the bias term? According to He et al., the biases are in the `BN` layers that immediately follow the convolutions [48], so there is no need to introduce a second bias term.

Next, we have our second `CONV` layer in the bottleneck, this one responsible for learning a total of $K/4$, 3×3 filters:

```
31          # the second block of the ResNet module are the 3x3 CONVs
32          bn2 = BatchNormalization(axis=chanDim, epsilon=bnEps,
33              momentum=bnMom)(conv1)
34          act2 = Activation("relu")(bn2)
35          conv2 = Conv2D(int(K * 0.25), (3, 3), strides=stride,
36              padding="same", use_bias=False,
37              kernel_regularizer=l2(reg))(act2)
```

The final block in the bottleneck learns K filters, each of which are 1×1:

```
39          # the third block of the ResNet module is another set of 1x1
40          # CONVs
41          bn3 = BatchNormalization(axis=chanDim, epsilon=bnEps,
42              momentum=bnMom)(conv2)
43          act3 = Activation("relu")(bn3)
44          conv3 = Conv2D(K, (1, 1), use_bias=False,
45              kernel_regularizer=l2(reg))(act3)
```

For more details on why we call this a "bottleneck" with "pre-activation", please see Section 12.1 above.

The next step is to see if we need to reduce spatial dimensions, thereby alleviating the need to apply max pooling:

```
47          # if we are to reduce the spatial size, apply a CONV layer to
48          # the shortcut
49          if red:
50              shortcut = Conv2D(K, (1, 1), strides=stride,
51                  use_bias=False, kernel_regularizer=l2(reg))(act1)
```

If we *are* instructed to reduce spatial dimensions, we'll do so with a convolutional layer (applied to the shortcut) with a stride > 1.

The output of the final `conv3` in the bottleneck is the added together with the `shortcut`, thus serving as the output of the `residual_module`:

```
53          # add together the shortcut and the final CONV
54          x = add([conv3, shortcut])
55
56          # return the addition as the output of the ResNet module
57          return x
```

The `residual_module` will serve as our building block when creating deep residual networks. Let's move on to using this building block inside the `build` method:

```
59      @staticmethod
60      def build(width, height, depth, classes, stages, filters,
61          reg=0.0001, bnEps=2e-5, bnMom=0.9, dataset="cifar"):
```

Just as our `residual_module` requires more parameters than previous micro-architecture implementations, the same is true for our `build` function. The `width`, `height`, and `depth` classes all control the input spatial dimensions of the images in our dataset. The `classes` variable dictates how many overall classes our network should learn – these variables you have already seen.

What is interesting are the `stages` and `filters` parameters, both of which are *lists*. When constructing the ResNet architecture, we'll be stacking a number of residual modules on top of each other (using the same number of filters for each stack), followed by reducing the spatial dimensions of the volume – this process is then continued until we are ready to apply our average pooling and softmax classifier.

To make this point clear, let's suppose that `stages=(3, 4, 6)` and `filters=(64, 128, 256, 512)`. The first filter value, 64, will be applied to the only `CONV` layer *not* part of the residual module (i.e., first convolutional layer in the network). We'll then stack *three* residual modules on top of each other – each of these residual modules will learn $K = 128$ filters. The spatial dimensions of the volume will be reduced, and then we'll move on to the second entry in `stages` where we'll stack *four* residual modules on top of each other, each responsible for learning $K = 256$ filters. After these four residual modules, we'll again reduce dimensionality and move on to the final entry in the `stages` list, instructing us to stack *six* residual modules on top of each other, where each residual module will learn $K = 512$.

The benefit of specifying both `stages` and `filters` in a list (rather than hardcoding them) is that we can easily leverage `for` loops to build the very deep network architectures without introducing code bloat – this point will become more clear later in our implementation.

Finally, we have the `dataset` parameter which is assumed to be a string. Depending on the dataset we are building ResNet for, we may want to apply more/less convolutions and batch normalizations *before* we start stacking our residual modules. We'll see why we might want to vary the number of convolutional layers in Section 12.5 below, but for the time being, you can safely ignore this parameter.

Next, let's initialize our `inputShape` and `chanDim` based on whether we are using "channels last" (**Lines 64 and 65**) or "channels first" (**Lines 69-71**) ordering.

```
62          # initialize the input shape to be "channels last" and the
63          # channels dimension itself
64          inputShape = (height, width, depth)
65          chanDim = -1
66
67          # if we are using "channels first", update the input shape
68          # and channels dimension
```

```
69          if K.image_data_format() == "channels_first":
70              inputShape = (depth, height, width)
71              chanDim = 1
```

We are now ready to define the `Input` to our `ResNet` implementation:

```
73          # set the input and apply BN
74          inputs = Input(shape=inputShape)
75          x = BatchNormalization(axis=chanDim, epsilon=bnEps,
76              momentum=bnMom)(inputs)
77
78          # check if we are utilizing the CIFAR dataset
79          if dataset == "cifar":
80              # apply a single CONV layer
81              x = Conv2D(filters[0], (3, 3), use_bias=False,
82                  padding="same", kernel_regularizer=l2(reg))(x)
```

Unlike previous network architectures we have seen in this book (where the first layer is typically a `CONV`), we see that ResNet uses a `BN` as the first layer. The reasoning behind applying batch normalization to your input is an added level of normalization. In fact, performing batch normalization on the input itself can sometimes remove the need to apply mean normalization to the inputs. In either case, the `BN` on **Lines 75 and 76** acts as an added level of normalization.

From there, we apply a single `CONV` layer on **Lines 81 and 82**. This `CONV` layer will learn a total of `filters[0]`, 3×3 filters (keep in mind that `filters` is a *list*, so this value is specified via the `build` method when constructing the architecture).

You'll also notice that I've made a check to see if we are using the CIFAR-10 dataset (**Line 79**). Later in this chapter, we'll be updating this `if` block to include an `elif` statement for Tiny ImageNet. Since the input dimensions to Tiny ImageNet are larger, we'll apply a series of convolutions, batch normalizations, and max pooling (the only max pooling in the ResNet architecture) before we start stacking residual modules. However, for the time being, we are only using the CIFAR-10 dataset.

Let's go ahead and start stacking residual layers on top of each other, the cornerstone of the ResNet architecture:

```
84          # loop over the number of stages
85          for i in range(0, len(stages)):
86              # initialize the stride, then apply a residual module
87              # used to reduce the spatial size of the input volume
88              stride = (1, 1) if i == 0 else (2, 2)
89              x = ResNet.residual_module(x, filters[i + 1], stride,
90                  chanDim, red=True, bnEps=bnEps, bnMom=bnMom)
91
92              # loop over the number of layers in the stage
93              for j in range(0, stages[i] - 1):
94                  # apply a ResNet module
95                  x = ResNet.residual_module(x, filters[i + 1],
96                      (1, 1), chanDim, bnEps=bnEps, bnMom=bnMom)
```

On **Line 85** we start looping over the list of `stages`. Keep in mind that every entry in the `stages` list is an *integer,* indicating how many residual modules will be stacked on top of each other. Following the work of Springenberg et al., ResNet tries to reduce the usage of pooling as much as possible, relying on `CONV` layers to reduce the spatial dimensions of a volume.

To reduce volume size without pooling layers, we must set the `stride` of the convolution on **Line 88**. If this is the *first* entry in the `stage`, we'll set the `stride` to `(1, 1)`, indicating that *no* downsampling should be performed. However, for every *subsequent* stage we'll apply a residual module with a `stride` of `(2, 2)`, which will allow us to decrease the volume size.

From there, we'll loop over the number of layers in the current stage on **Line 93** (i.e., the number of residual modules that will be stacked on top of each other). The number of filters each residual module will learn is controlled by the corresponding entry in the `filters` list. The reason we use `i + 1` as the index into `filters` is because the *first* filter value was used on **Lines 81 and 82**. The rest of the filter values correspond to the number of filters in each stage. Once we have stacked `stages[i]` residual modules on top of each other, our `for` loop brings us back up to **Lines 88-90** where we decrease the spatial dimensions of the volume and repeat the process.

At this point, our volume size has been reduced to `8 x 8 x num_filters` (you can verify this for yourself by computing the input/output volume sizes for each layer, or better yet, simply using the `plot_model` function from Chapter 19 of the *Starter Bundle*).

In order to avoid using dense fully-connected layers, we'll instead apply average pooling to reduce the volume size to `1 x 1 x classes`:

```
98          # apply BN => ACT => POOL
99          x = BatchNormalization(axis=chanDim, epsilon=bnEps,
100             momentum=bnMom)(x)
101         x = Activation("relu")(x)
102         x = AveragePooling2D((8, 8))(x)
```

From there, we create a dense layer for the total number of `classes` we are going to learn, followed by applying a softmax activation to obtain our final output probabilities:

```
104         # softmax classifier
105         x = Flatten()(x)
106         x = Dense(classes, kernel_regularizer=l2(reg))(x)
107         x = Activation("softmax")(x)
108
109         # create the model
110         model = Model(inputs, x, name="resnet")
111
112         # return the constructed network architecture
113         return model
```

The fully constructed ResNet `model` is then returned to the calling function on **Line 113**.

12.3 ResNet on CIFAR-10

Outside of training smaller variants of ResNet on the *full* ImageNet dataset, I had never attempted to train ResNet on CIFAR-10 (or Stanford's Tiny ImageNet challenge, as we'll see in this section). Because of this fact, I have decided to treat this section and the next as candid case studies where I reveal my personal rules of thumb and best practices I have developed over *years* of training neural networks.

These best practices allow me to approach a new problem with an initial plan, iterate on it, and eventually arrive at a solution that obtains good accuracy. In the case of CIFAR-10, we'll be able to replicate the performance of He et al. and claim a spot amongst other state-of-the-art approaches [49].

12.3.1 Training ResNet on CIFAR-10 With the ctrl + c Method

Whenever I start a new set of experiments with either a network architecture I am unfamiliar with, a dataset I have never worked with, or both, I always begin with the `ctrl + c` method of training. Using this method, I can start training with an initial learning rate (and associated set of hyperparameters), monitor training, and quickly adjust the learning rate based on the results as they come in. This method is *especially helpful* when I am totally unsure on the approximate number of epochs it will take for a given architecture to obtain reasonable accuracy or a specific dataset.

In the case of CIFAR-10, I have previous experience (as do you, after reading all the other chapters in this book), so I'm quite confident that it will take 60-100 epochs, but I'm not exactly sure since I've never trained ResNet on the CIFAR-10 before.

Therefore, our first few experiments will rely on the `ctrl + c` method of training to narrow in on what hyperparameters we should be using. Once we are comfortable with our set of hyperparameters, we'll switch over to a *specific* learning rate decay schedule in hopes of milking every last bit of accuracy out of the training process.

To get started, open up a new file, name it `resnet_cifar10.py`, and insert the following code:

```
1  # set the matplotlib backend so figures can be saved in the background
2  import matplotlib
3  matplotlib.use("Agg")
4
5  # import the necessary packages
6  from sklearn.preprocessing import LabelBinarizer
7  from pyimagesearch.nn.conv import ResNet
8  from pyimagesearch.callbacks import EpochCheckpoint
9  from pyimagesearch.callbacks import TrainingMonitor
10 from tensorflow.keras.preprocessing.image import ImageDataGenerator
11 from tensorflow.keras.optimizers import SGD
12 from tensorflow.keras.datasets import cifar10
13 from tensorflow.keras.models import load_model
14 import tensorflow.keras.backend as K
15 import numpy as np
16 import argparse
17 import sys
18
19 # set a high recursion limit so Theano doesn't complain
20 sys.setrecursionlimit(5000)
```

We start by importing our required Python packages on **Lines 6-17**. Since we'll be using the `ctrl + c` method to training, we'll make sure to import the `EpochCheckpoint` class (**Line 8**) to serialize ResNet weights to disk during the training process, allowing us to stop and restart training from a specific checkpoint. Since this is our first experiment with ResNet, we'll be using the SGD optimizer (**Line 11**) – time will tell if we decide to switch and use a different optimizer (we'll let our results dictate that).

On **Line 20** I update the recursion limit for the Python programming language. I wrote this book with *both* TensorFlow and Theano in mind, so if you are using TensorFlow, you don't have to worry about this line. However, if you are using Theano, you may encounter an error when instantiating the `ResNet` architecture that a maximum recursion level has been reached. This is a known "bug" with Theano and can be resolved simply by increasing the recursion limit of the Python programming language [50].

Next, let's parse our command line arguments:

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

Our script will require only the `--checkpoints` switch, the path to the directory where we will store the ResNet weights every *N* epochs. In the case that we need to restart training from a particular epoch, we can supply the `--model` path along with an integer indicating the *specific* epoch number.

The next step is to load the CIFAR-10 dataset from disk (pre-split into training and testing), perform mean subtraction, and one-hot encode the integer labels as vectors:

```python
# load the training and testing data, converting the images from
# integers to floats
print("[INFO] loading CIFAR-10 data...")
((trainX, trainY), (testX, testY)) = cifar10.load_data()
trainX = trainX.astype("float")
testX = testX.astype("float")

# apply mean subtraction to the data
mean = np.mean(trainX, axis=0)
trainX -= mean
testX -= mean

# convert the labels from integers to vectors
lb = LabelBinarizer()
trainY = lb.fit_transform(trainY)
testY = lb.transform(testY)
```

While we're at it, let's also initialize an `ImageDataGenerator` so we can apply data augmentation to CIFAR-10:

```python
# construct the image generator for data augmentation
aug = ImageDataGenerator(width_shift_range=0.1,
    height_shift_range=0.1, horizontal_flip=True,
    fill_mode="nearest")
```

In the case we are training ResNet from the very first epoch, we need to instantiate the network architecture:

```python
# if there is no specific model checkpoint supplied, then initialize
# the network (ResNet-56) and compile the model
if args["model"] is None:
    print("[INFO] compiling model...")
    opt = SGD(lr=1e-1)
```

```
59      model = ResNet.build(32, 32, 3, 10, (9, 9, 9),
60          (64, 64, 128, 256), reg=0.0005)
61      model.compile(loss="categorical_crossentropy", optimizer=opt,
62          metrics=["accuracy"])
```

To start, take a look at the learning rate for our SGD optimizer on **Line 58** – at $1e-1$ this learning rate is by far the largest we have used in this book (by an order of magnitude). The reason we are able to get away with such a high learning rate is due to the identity mappings built into the residual module. Learning rates this high would not (typically) work for networks such as AlexNet, VGG, etc.

We then instantiate our ResNet model on **Lines 59 and 60**. Here we can see that the network will accept input images with a width of 32 pixels, height of 32 pixels, and depth of 3 (one for each of the RGB channels in the CIFAR-10 dataset). Since the CIFAR-10 dataset has ten classes, we'll learn ten output labels.

The next parameter we need to supply is (9, 9, 9), or the number of stages in our architecture. This tuple indicates that we will be learning three stages with each stage containing nine residual modules stacked on top of each other. In between each stage, we will apply an additional residual module to decrease the volume size.

The next parameter, (64, 64, 128, 256) is the number of filters that the CONV layers will learn. The first CONV layer (before any residual model is applied) will learn $K = 64$ filters. The remaining entries, 64, 128, and 256 correspond to the number of filters each of the residual module stages will learn. For example, the first nine residual modules will learn $K = 64$ filters. The second set of nine residual modules will learn $K = 128$ filters. And finally, the last set of nine residual modules will learn $K = 256$ filters. The last argument we'll supply to ResNet is reg, or our L2 regularization strength for weight decay – this value is crucial as it will enable us to prevent overfitting.

In the case we start restarting training from a specific epoch, we need to load the network weights from disk and update the learning rate:

```
64  # otherwise, load the checkpoint from disk
65  else:
66      print("[INFO] loading {}...".format(args["model"]))
67      model = load_model(args["model"])
68
69      # update the learning rate
70      print("[INFO] old learning rate: {}".format(
71          K.get_value(model.optimizer.lr)))
72      K.set_value(model.optimizer.lr, 1e-5)
73      print("[INFO] new learning rate: {}".format(
74          K.get_value(model.optimizer.lr)))
```

Let's also construct a set of callbacks so we can both (1) checkpoint ResNet weights every five epochs and (2) monitor training:

```
76  # construct the set of callbacks
77  callbacks = [
78      EpochCheckpoint(args["checkpoints"], every=5,
79          startAt=args["start_epoch"]),
80      TrainingMonitor("output/resnet56_cifar10.png",
81          jsonPath="output/resnet56_cifar10.json",
82          startAt=args["start_epoch"])]
```

Finally, we'll train our network in batch sizes of 128:

```python
# train the network
print("[INFO] training network...")
model.fit_generator(
	aug.flow(trainX, trainY, batch_size=128),
	validation_data=(testX, testY),
	steps_per_epoch=len(trainX) // 128, epochs=100,
	callbacks=callbacks, verbose=1)
```

Now that our `resnet_cifar10.py` script is coded up, let's move on to running experiments with it.

ResNet on CIFAR-10: Experiment #1

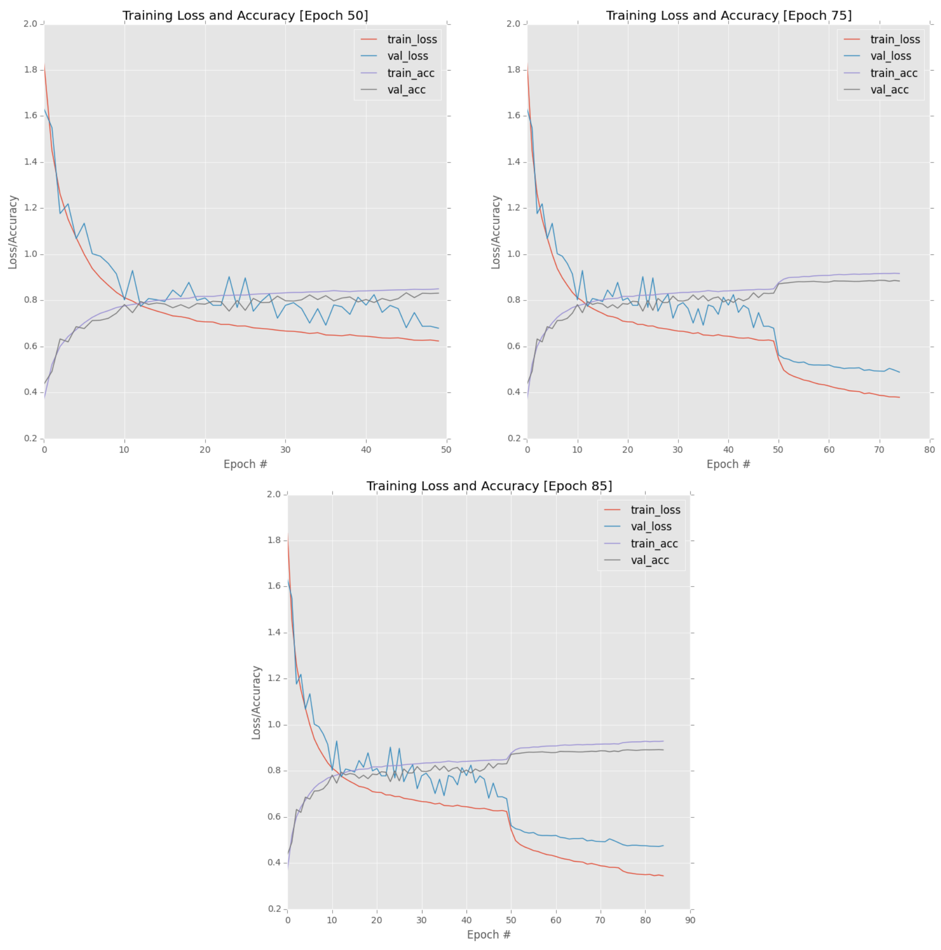


Figure 12.4: **Top-left:** First 50 epochs when training ResNet on CIFAR-10 in Experiment #1. **Top-right:** Next 25 epochs. **Bottom:** Final 10 epochs.

In my very first experiment with CIFAR-10, I was worried about the number of filters in the network, especially regarding overfitting. Because of this concern, my initial filter list consisted of `(16, 16, 32, 64)` along with `(9, 9, 9)` stages of residual modules. I also applied a very small amount of L2 regularization with `reg=0.0001` – I knew regularization would be needed, but I wasn't sure on the correct amount (yet). ResNet was trained using SGD with a base learning rate of $1e-1$ and a momentum term of 0.9.

I started training using the following command:

```
$ python resnet_cifar10.py --checkpoints output/checkpoints
```

Past epoch 50 I noticed training loss starting to slow as well as some volatility in the validation loss (and a growing gap between the two) (Figure 12.4, *top-left*). I stopped training, lowered the learning rate to $1e-2$, and then continued training:

```
$ python resnet_cifar10.py --checkpoints output/checkpoints \
     --model output/checkpoints/epoch_50.hdf5 --start-epoch 50
```

The drop in learning rate proved very effective, stabilizing validation loss, but also overfitting on the training set start to creep in (in inevitability when working with CIFAR-10) around epoch 75 (Figure 12.4, *top-right*). After epoch 75 I once again stopped training, lowered the learning rate to $1e-3$, and allowed ResNet to continue training for another 10 epochs:

```
$ python resnet_cifar10.py --checkpoints output/checkpoints \
     --model output/checkpoints/epoch_75.hdf5 --start-epoch 75
```

The final plot is shown in Figure 12.4 (*bottom*), where we reach 89.06% accuracy on the validation set. For our very first experiment 89.06% is a good start; however, it's not as high as the 90.81% achieved by GoogLeNet in Chapter 11. Furthermore, He et al. reported an accuracy of 93% with ResNet on CIFAR-10, so we clearly have some work to do.

12.3.2 ResNet on CIFAR-10: Experiment #2

Our previous experiment achieved a reasonable accuracy of 89.06% accuracy – but we need higher accuracy. Instead of increasing the depth of the network (by adding more stages), I decided to add more *filters* to each of the `CONV` layers. Thus, my `filters` list was updated to be `(16, 64, 128, 256)`.

Notice how the number of filters in all residual modules have *doubled* from the previous experiment (the number of filters in the first `CONV` layer was left the same). SGD was once again used to train the network with a momentum term of 0.9. I also kept the regularization term at 0.0001.

In Figure 12.5 (*top-left*) you can find a plot of my first 40 epochs: We can clearly see a gap between training loss and validation loss, but overall, validation accuracy is still keeping up with training accuracy. In an effort to improve the accuracy, I decided to lower the learning rate from $1e-1$ to $1e-2$ and train for another five epochs – the result was that learning stagnated entirely (*top-right*). Lowering learning rate again from $1e-2$ to $1e-3$ even caused overfitting through a slight rise in validation loss (*bottom*)

Interestingly, the loss stagnated for *both* the training set *and* the validation set, not unlike previous experiments we've run with GoogLeNet. After the initial drop in learning rate, it appears that our network could not learn any more underlying patterns in the dataset. All that said, after the 50th epoch validation accuracy had increased to **90.10%**, an improvement from our first experiment.

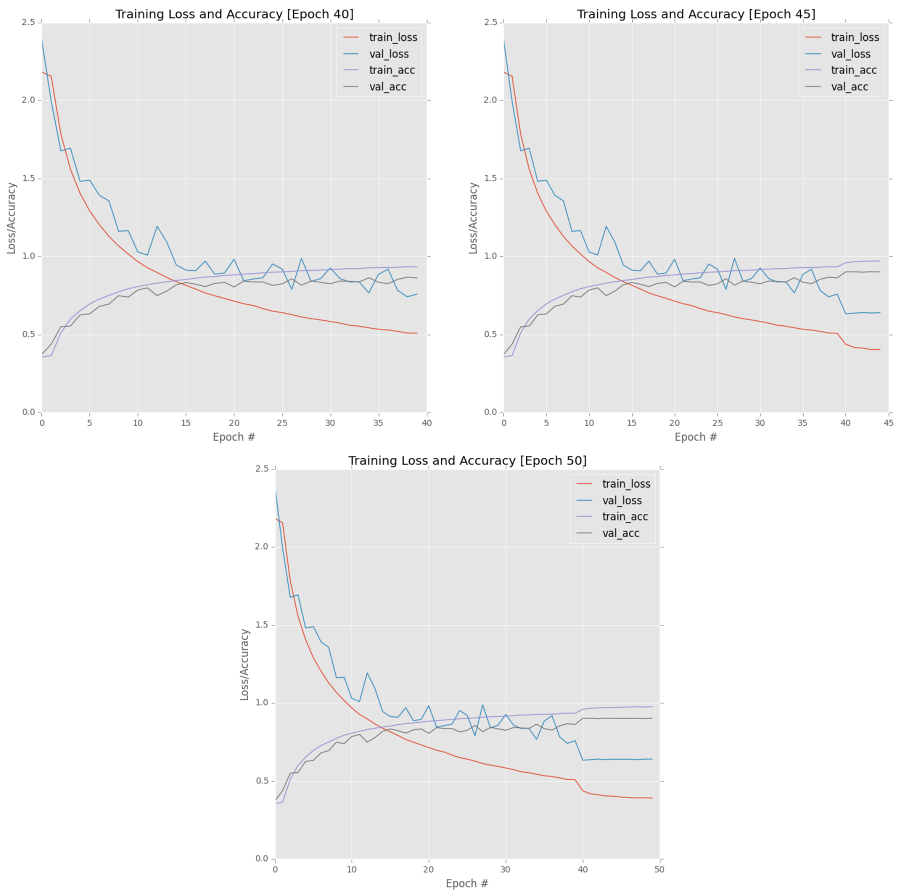


Figure 12.5: **Top-left:** First 40 epochs when training ResNet on CIFAR-10 in Experiment #2. **Top-right:** Next 5 epochs. **Bottom:** Final 5 epochs.

ResNet on CIFAR-10: Experiment #3

At this point, I was starting to become more comfortable training ResNet on CIFAR-10. Clearly the increase of filters helped, but the stagnation in learning after the first learning rate drop was still troubling. I was confident that a slow, linear decrease in learning rate would help combat this problem, but I wasn't convinced that I had obtained a good set of hyperparameters to warrant switching over to learning rate decay.

Instead, I decided to increase the number of filters learned in the *first* `CONV` layer to 64 (up from 16), turning the `filters` list into `(64, 64, 128, 256)`. The increase in filters helped in the second experiment, and there is no reason the first `CONV` layer should miss out on these benefits as well. The SGD optimizer was left alone with an initial learning rate of $1e-1$ and momentum of 0.9.

Furthermore, I also decided to dramatically increase regularization from 0.0001 to 0.0005. I had a suspicion that allowing the network to train for *longer* would result in higher validation accuracy – using a larger regularization term would likely enable me to train for longer.

I was also considering lowering my learning rate, but given that the network was making

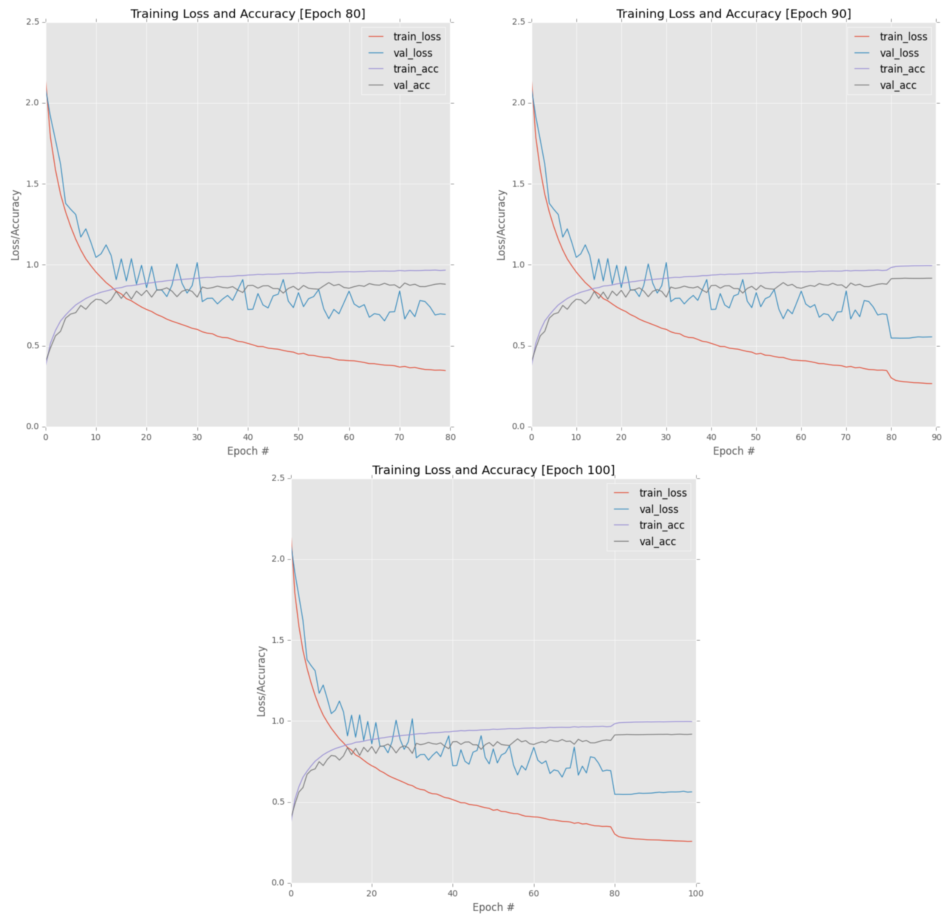


Figure 12.6: **Top-left:** First 80 epochs when training ResNet on CIFAR-10 in Experiment #3. **Top-right:** Next 10 epochs. **Bottom:** Final 10 epochs.

traction without a problem at $1e-1$, it hardly seemed worth it to lower to $1e-2$. Doing so might have stabilized training (i.e., less fluctuation in validation loss/accuracy), but would have ultimately led to lower accuracy after training completed. If my larger learning rate + larger regularization term suspicion turned out to be correct, then it would make sense to switch over to a learning rate decay to avoid stagnation after order of magnitude drops. But before I could make this switch, I first needed to prove my hunch.

As my plot of the first 80 epochs demonstrates (Figure 12.6, *top-left*), there certainly is overfitting as validation quickly diverges from training. However, what's interesting here is that while overfitting is undoubtedly occurring (as training loss drops much faster than validation loss), we are able to train the network for *longer* without validation loss starting to *increase*.

After the 80th epoch, I stopped training, lowered the learning rate to $1e-2$, then trained for another 10 epochs (Figure 12.6, *top-right*). We see an initial drop and loss and increase in accuracy, but from there validation loss/accuracy plateaus. Furthermore, we can start to see the validation loss *increase*, a sure sign of overfitting.

To validate that overfitting was indeed happening, I stopped training at epoch 90, lowered the learning rate to 1e-3, then trained for another 10 epochs (Figure 12.6, *bottom*). Sure enough, this is the telltale sign of overfitting: validation loss *increasing* while training loss decreases/remains constant.

However, what's very interesting is that after the 100th epoch we obtained **91.83%** validation accuracy, higher than our second experiment. The downside is that we are overfit – we need a way to maintain this level of accuracy (and increase it) without overfitting. To do so, I decided to switch from `ctrl + c` training to learning rate decay.

12.4 Training ResNet on CIFAR-10 with Learning Rate Decay

At this point, it seems that we have gotten as far as we can using standard `ctrl + c` training. We've also been able to see that our most successful experiments occur when we can train for longer, in the range of 80-100 epochs. However, there are two major problems we need to overcome:

1. Whenever we drop the learning rate by an order of magnitude and restart training, we obtain a nice bump in accuracy, but then we quickly plateau.
2. We are overfitting.

To solve these problems, and boost accuracy further, a good experiment to try is linearly decreasing the learning rate over a large number of epochs, typically about the same as your *longest* `ctrl + c` experiments (if not slightly longer). To start this, let's open up a new file, name it `resnet_cifar10_decay.py`, and insert the following code:

```
1  # set the matplotlib backend so figures can be saved in the background
2  import matplotlib
3  matplotlib.use("Agg")
4
5  # import the necessary packages
6  from sklearn.preprocessing import LabelBinarizer
7  from pyimagesearch.nn.conv import ResNet
8  from pyimagesearch.callbacks import TrainingMonitor
9  from tensorflow.keras.preprocessing.image import ImageDataGenerator
10 from tensorflow.keras.callbacks import LearningRateScheduler
11 from tensorflow.keras.optimizers import SGD
12 from tensorflow.keras.datasets import cifar10
13 import numpy as np
14 import argparse
15 import sys
16 import os
17
18 # set a high recursion limit so Theano doesn't complain
19 sys.setrecursionlimit(5000)
```

Line 2 configures matplotlib so we can save plots in the background. We then import the remainder of our Python packages on **Lines 6-16**. Take a look at **Line 10** where we import our `LearningRateScheduler` so we can define a custom learning rate decay for the training process. We then set a high system recursion limit in order to avoid any issues with the Theano backend (just in case you are using it).

The next step is to define the learning rate decay schedule:

```
21 # define the total number of epochs to train for along with the
22 # initial learning rate
```

```
NUM_EPOCHS = 100
INIT_LR = 1e-1

def poly_decay(epoch):
    # initialize the maximum number of epochs, base learning rate,
    # and power of the polynomial
    maxEpochs = NUM_EPOCHS
    baseLR = INIT_LR
    power = 1.0

    # compute the new learning rate based on polynomial decay
    alpha = baseLR * (1 - (epoch / float(maxEpochs))) ** power

    # return the new learning rate
    return alpha
```

We'll train our network for a total of 100 epochs with a base learning rate of $1e-1$. The `poly_decay` function will decay our $1e-1$ learning rate *linearly* over the course of 100 epochs. This is a *linear* decay due to the fact that we set `power=1`. For more information on learning rate schedules, see Chapter 16 of the *Starter Bundle* along with Chapter 11 of the *Practitioner Bundle*.

We then need to supply two command line arguments:

```
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-m", "--model", required=True,
    help="path to output model")
ap.add_argument("-o", "--output", required=True,
    help="path to output directory (logs, plots, etc.)")
args = vars(ap.parse_args())
```

The `--model` switch controls the path to our final serialized model after training, while `--output` is the base directory to where we will store any logs, plots, etc.

We can now load the CIFAR-10 dataset and mean normalize it:

```
# load the training and testing data, converting the images from
# integers to floats
print("[INFO] loading CIFAR-10 data...")
((trainX, trainY), (testX, testY)) = cifar10.load_data()
trainX = trainX.astype("float")
testX = testX.astype("float")

# apply mean subtraction to the data
mean = np.mean(trainX, axis=0)
trainX -= mean
testX -= mean
```

Encode the integer labels as vectors:

```
# convert the labels from integers to vectors
lb = LabelBinarizer()
trainY = lb.fit_transform(trainY)
testY = lb.transform(testY)
```

As well as initialize our ImageDataGenerator for data argumentation:

```python
# construct the image generator for data augmentation
aug = ImageDataGenerator(width_shift_range=0.1,
     height_shift_range=0.1, horizontal_flip=True,
     fill_mode="nearest")
```

Our callbacks list will consist of both a TrainingMonitor along with a LearningRateScheduler with the poly_decay function supplied as the only argument – this class will allow us to decay our learning rate as we train.

```python
# construct the set of callbacks
figPath = os.path.sep.join([args["output"], "{}.png".format(
     os.getpid())])
jsonPath = os.path.sep.join([args["output"], "{}.json".format(
     os.getpid())])
callbacks = [TrainingMonitor(figPath, jsonPath=jsonPath),
     LearningRateScheduler(poly_decay)]
```

We'll then instantiate ResNet with the best parameters we found from Section 12.6 (three stacks of nine residual modules: 64 filters in the first CONV layer before the residual modules, and 64, 128, and 256 filters for each stack of respective residual modules):

```python
# initialize the optimizer and model (ResNet-56)
print("[INFO] compiling model...")
opt = SGD(lr=INIT_LR, momentum=0.9)
model = ResNet.build(32, 32, 3, 10, (9, 9, 9),
     (64, 64, 128, 256), reg=0.0005)
model.compile(loss="categorical_crossentropy", optimizer=opt,
     metrics=["accuracy"])
```

We'll then train our network using learning rate decay:

```python
# train the network
print("[INFO] training network...")
model.fit_generator(
     aug.flow(trainX, trainY, batch_size=128),
     validation_data=(testX, testY),
     steps_per_epoch=len(trainX) // 128, epochs=NUM_EPOCHS,
     callbacks=callbacks, verbose=1)

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

The big question is – will our learning rate decay pay off? To find out, proceed to the next section.

ResNet on CIFAR-10: Experiment #4

As the code in the previous section indicates, we are going to use the SGD optimizer with a base learning rate of $1e-1$ and a momentum term of 0.9. We'll train ResNet for a total of 100 epochs, linearly decreasing the linear rate from $1e-1$ down to zero. To train ResNet on CIFAR-10 with learning rate decay, I executed the following command:

```
$ python resnet_cifar10_decay.py --output output \
     --model output/resnet_cifar10.hdf5
```

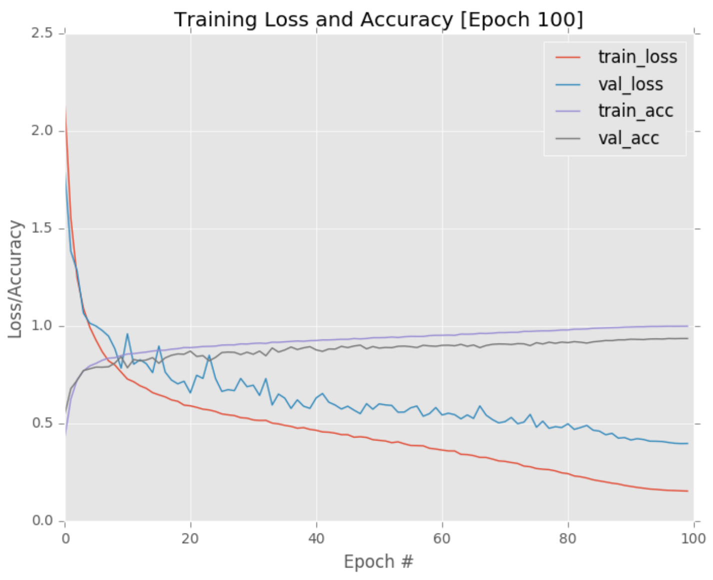


Figure 12.7: Training ResNet on CIFAR-10 using learning rate decay beats out all previous experiments.

After training was complete, I took a look at the plot (Figure 12.7). As in previous experiments, training and validation loss start to diverge early on, but more importantly, *the gap remains approximately constant* after the initial divergence. This result is important as it indicates that our overfitting is *controlled*. We have to accept that we *will* overfit when training on CIFAR-10, but we need to *control* this overfitting. By applying learning rate decay, we were able to successfully do so.

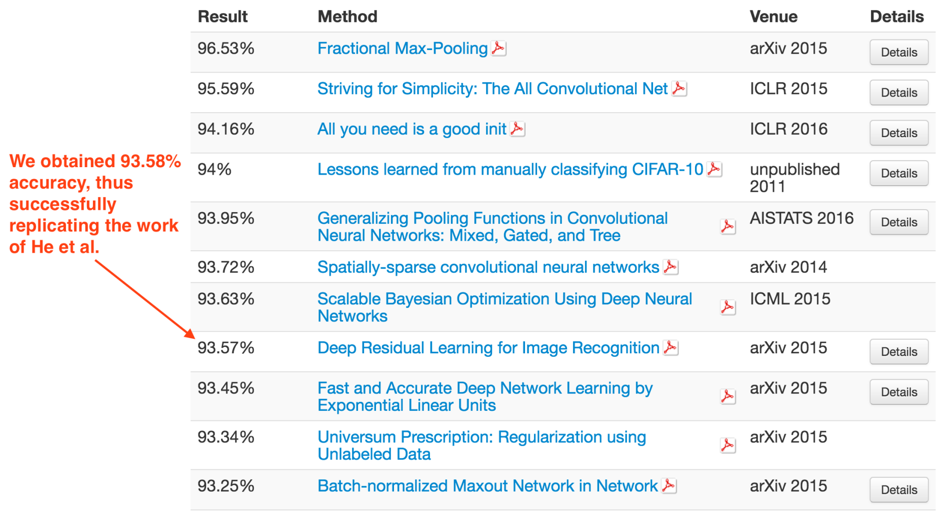
The question is, *did we obtain higher classification accuracy?* To answer that, take a look at the output of the last few epochs:

```
...
Epoch 98/100
247s - loss: 0.1563 - acc: 0.9985 - val_loss: 0.3987 - val_acc: 0.9351
Epoch 99/100
245s - loss: 0.1548 - acc: 0.9987 - val_loss: 0.3973 - val_acc: 0.9358
Epoch 100/100
```

```
244s - loss: 0.1538 - acc: 0.9990 - val_loss: 0.3978 - val_acc: 0.9358
[INFO] serializing network...
```

After the 100th epoch, ResNet is reaching **93.58%** accuracy on our testing set. This result is *substantially* higher than our previous two experiments, and more importantly, it has allowed us to replicate the results from He et al. when training ResNet on CIFAR-10.

Taking a look at the CIFAR-10 leaderboard [49], we see that He et al. reached 93.57% accuracy [24], near identical to our result (Figure 12.8). The red arrow indicates our accuracy, safely landing us in the top-10 leaderboard.



Result	Method	Venue	Details
96.53%	Fractional Max-Pooling	arXiv 2015	Details
95.59%	Striving for Simplicity: The All Convolutional Net	ICLR 2015	Details
94.16%	All you need is a good init	ICLR 2016	Details
94%	Lessons learned from manually classifying CIFAR-10	unpublished 2011	Details
93.95%	Generalizing Pooling Functions in Convolutional Neural Networks: Mixed, Gated, and Tree	AISTATS 2016	Details
93.72%	Spatially-sparse convolutional neural networks	arXiv 2014	
93.63%	Scalable Bayesian Optimization Using Deep Neural Networks	ICML 2015	
93.57%	Deep Residual Learning for Image Recognition	arXiv 2015	Details
93.45%	Fast and Accurate Deep Network Learning by Exponential Linear Units	arXiv 2015	Details
93.34%	Universum Prescription: Regularization using Unlabeled Data	arXiv 2015	
93.25%	Batch-normalized Maxout Network in Network	arXiv 2015	Details

We obtained 93.58% accuracy, thus successfully replicating the work of He et al.

Figure 12.8: We have successfully replicated the work of He et al. when applying ResNet to CIFAR-10 down to 0.01%.

12.5 ResNet on Tiny ImageNet

In this section, we will train the ResNet architecture (with bottleneck and pre-activation) on Stanford's cs231n Tiny ImageNet challenge. Similar to the ResNet + CIFAR-10 experiments earlier in this chapter, I have never trained ResNet on Tiny ImageNet before, so I'm going to apply my same exact experiment process:

1. Start with `ctrl + c`-based training to obtain a baseline.
2. If stagnation/plateauing occurs after order of magnitude learning rate drops, then switch over to learning rate decay.

Given that we've already applied a similar technique to CIFAR-10, we should be able to save ourselves some time noticing signs of overfitting and plateauing earlier. That said, let's get started by reviewing directory structure for this project, which is near identical to the GoogLeNet and Tiny ImageNet challenge from the previous chapter:

```
|--- resnet_tinyimagenet.py
|    |--- config
|    |    |--- __init__.py
|     |     |--- tiny_imagenet_config.py
|     |--- rank_accuracy.py
|     |--- train.py
|     |--- train_decay.py
|     |--- output/
|     |     |--- checkpoints/
|     |     |--- tiny-image-net-200-mean.json
```

Here you can see we have created `config` Python module where we have stored a file named `tiny_imagenet_config.py`.

This file was copied directly from the GoogLeNet chapter. I then updated **Lines 31-39** to point to the output ResNet `MODEL_PATH`, `FIG_PATH` (learning plot), and `JSON_PATH` (serialized log of training history):

```
31 # define the path to the output directory used for storing plots,
32 # classification reports, etc.
33 OUTPUT_PATH = "output"
34 MODEL_PATH = path.sep.join([OUTPUT_PATH,
35      "resnet_tinyimagenet.hdf5"])
36 FIG_PATH = path.sep.join([OUTPUT_PATH,
37      "resnet56_tinyimagenet.png"])
38 JSON_PATH = path.sep.join([OUTPUT_PATH,
39      "resnet56_tinyimagenet.json"])
```

From there we have `train.py` which will be responsible for training ResNet using the standard `ctrl + c` method. In the case that we wish to apply learning rate decay, we'll be able to use `train_decay.py`. Finally, `rank_accuracy.py` will be used to compute the rank-1 and rank-5 accuracy of ResNet on Tiny ImageNet.

Additionally, do not forget to copy your `tiny-image-net-200-mean.json` file from Chapter 11 to the `output` directory for the ResNet project. You will need the mean file when performing mean subtraction during training and evaluation.

12.5.1 Updating the ResNet Architecture

Earlier in this chapter we reviewed our implementation of the `ResNet` architecture in detail. Specifically, we noted the `dataset` parameter supplied to the `build` method, like so:

```
     @staticmethod
     def build(width, height, depth, classes, stages, filters,
          reg=0.0001, bnEps=2e-5, bnMom=0.9, dataset="cifar"):
```

This value defaulted to `cifar`; however, since we are now working with Tiny ImageNet, we need to update our `ResNet` implementation to include an `if/elif` block. Go ahead and open up your `resnet.py` file in the `nn.conv` sub-module of `PyImageSearch` and insert the following code:

```
          # check if we are utilizing the CIFAR dataset
          if dataset == "cifar":
               # apply a single CONV layer
               x = Conv2D(filters[0], (3, 3), use_bias=False,
                    padding="same", kernel_regularizer=l2(reg))(x)
```

```
83
84          # check to see if we are using the Tiny ImageNet dataset
85          elif dataset == "tiny_imagenet":
86              # apply CONV => BN => ACT => POOL to reduce spatial size
87              x = Conv2D(filters[0], (5, 5), use_bias=False,
88                  padding="same", kernel_regularizer=l2(reg))(x)
89              x = BatchNormalization(axis=chanDim, epsilon=bnEps,
90                  momentum=bnMom)(x)
91              x = Activation("relu")(x)
92              x = ZeroPadding2D((1, 1))(x)
93              x = MaxPooling2D((3, 3), strides=(2, 2))(x)
```

Lines 79-82 we have already reviewed before – these lines are where we apply a single 3×3 CONV layer for CIFAR-10. However, we are now updating the architecture to check for Tiny ImageNet (**Line 85**).

Provided we are instantiating ResNet for Tiny ImageNet, we need to add in some additional layers. To start, we apply 5×5 CONV layer to learn larger feature maps (in the *full* ImageNet dataset implementation, we'll actually be learning 7×7 filters).

Next, we apply a batch normalization followed a ReLU activation. Max pooling, the *only* max pooling layer in the ResNet architecture, is applied on **Line 93** using a size of 3×3 and stride of 2×2. Combined with the previous zero padding layer (**Line 92**), pooling ensures that our output spatial volume size is 32×32, the exact same spatial dimensions as the input images from CIFAR-10. Validating that the output volume size is 32×32 ensures we can easily reuse the rest of the ResNet implementation without having to make any additional changes.

12.5.2 Training ResNet on Tiny ImageNet With the ctrl + c Method

Now that our ResNet implementation has been updated, let's code up a Python script responsible for the actual training process. Open up a new file, name it `train.py`, and insert the following code:

```
# set the matplotlib backend so figures can be saved in the background
import matplotlib
matplotlib.use("Agg")

# import the necessary packages
from config import tiny_imagenet_config as config
from pyimagesearch.preprocessing import ImageToArrayPreprocessor
from pyimagesearch.preprocessing import SimplePreprocessor
from pyimagesearch.preprocessing import MeanPreprocessor
from pyimagesearch.callbacks import EpochCheckpoint
from pyimagesearch.callbacks import TrainingMonitor
from pyimagesearch.io import HDF5DatasetGenerator
from pyimagesearch.nn.conv import ResNet
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.models import load_model
import tensorflow.keras.backend as K
import argparse
import json
import sys

# set a high recursion limit so Theano doesn't complain
sys.setrecursionlimit(5000)
```

Lines 2 and 3 configure matplotlib so we can save our figures and plots to disk during the training process. We then import the remainder of our Python packages on **Lines 6-20**. We have seen all of these imports before from Chapter 11 on GoogLeNet + Tiny ImageNet, only now we are importing ResNet (**Line 13**) rather than GoogLeNet. We'll also update the maximum recursion limit just in case you are using the Theano backend.

Next comes our command line arguments:

```
25 # construct the argument parse and parse the arguments
26 ap = argparse.ArgumentParser()
27 ap.add_argument("-c", "--checkpoints", required=True,
28      help="path to output checkpoint directory")
29 ap.add_argument("-m", "--model", type=str,
30      help="path to *specific* model checkpoint to load")
31 ap.add_argument("-s", "--start-epoch", type=int, default=0,
32      help="epoch to restart training at")
33 args = vars(ap.parse_args())
```

These command line switches can be used to start training from scratch *or* restart training from a specific epoch. For a more detailed review of each command line argument, please see Section 12.3.1.

Let's also initialize our `ImageDataGenerator` that will be applied to the training set for data augmentation:

```
35 # construct the training image generator for data augmentation
36 aug = ImageDataGenerator(rotation_range=18, zoom_range=0.15,
37      width_shift_range=0.2, height_shift_range=0.2, shear_range=0.15,
38      horizontal_flip=True, fill_mode="nearest")
39
40 # load the RGB means for the training set
41 means = json.loads(open(config.DATASET_MEAN).read())
```

Line 41 then loads our RGB means computed over the training set of mean subtraction.

Our image pre-processors are fairly standard here, consisting of ensuring the image is resized to 64×64 pixels, performing mean normalization, and then converting the image to a Keras-compatible array:

```
43 # initialize the image preprocessors
44 sp = SimplePreprocessor(64, 64)
45 mp = MeanPreprocessor(means["R"], means["G"], means["B"])
46 iap = ImageToArrayPreprocessor()
47
48 # initialize the training and validation dataset generators
49 trainGen = HDF5DatasetGenerator(config.TRAIN_HDF5, 64, aug=aug,
50      preprocessors=[sp, mp, iap], classes=config.NUM_CLASSES)
51 valGen = HDF5DatasetGenerator(config.VAL_HDF5, 64,
52      preprocessors=[sp, mp, iap], classes=config.NUM_CLASSES)
```

Lines 49-52 construct an `HDF5DatasetGenerator`, supplying the image pre-processors and data augmentation classes. Batches of 64 images will be polled at a time for these generators and passed through the network.

If we are training ResNet from the very first epoch, we need to instantiate the model and compile it:

```
54 # if there is no specific model checkpoint supplied, then initialize
55 # the network and compile the model
56 if args["model"] is None:
57     print("[INFO] compiling model...")
58     model = ResNet.build(64, 64, 3, config.NUM_CLASSES, (3, 4, 6),
59         (64, 128, 256, 512), reg=0.0005, dataset="tiny_imagenet")
60     opt = SGD(lr=1e-1, momentum=0.9)
61     model.compile(loss="categorical_crossentropy", optimizer=opt,
62         metrics=["accuracy"])
```

Lines 58 and 59 initialize the `ResNet` model itself. The `model` will accept input images with $64 \times 64 \times 3$ spatial dimensions and will learn a total of `NUM_CLASSES`, which in the case of Tiny ImageNet is 200. My choice for `stages=(3, 4, 6)` was inspired by the *Deep Residual Learning for Image Recognition* [24] paper where a similar version of residual module layer stacking was used for the full ImageNet dataset.

Given that Tiny ImageNet will require more discriminative filters than CIFAR-10, I also updated the `filters` list to learn more filters for each of the `CONV` layers: `(64, 128, 256, 512)`. This list of filters implies that the *first* `CONV` layer (before any of the residual modules) will learn a total of 64 5×5 filters. From there, three residual modules will be stacked on top of each other, each responsible for learning $K = 128$ filters. Dimensionality is then reduced, then four residual modules are stacked, this time learning $K = 256$ filters. Once again the spatial dimensions of the volume are reduced, then six residual modules are stacked, each module learning $K = 512$ filters.

We'll also apply a regularization strength of 0.0005 as regularization seemed to aide us in training ResNet on CIFAR-10. To train the network, SGD will be used with a base learning rate of $1e-1$ and a momentum term of 0.9.

If we have stopped training, updated any hyperparameters (such as the learning rate), and wish to restart training, the following code block will handle that process for us:

```
64 # otherwise, load the checkpoint from disk
65 else:
66     print("[INFO] loading {}...".format(args["model"]))
67     model = load_model(args["model"])
68 
69     # update the learning rate
70     print("[INFO] old learning rate: {}".format(
71         K.get_value(model.optimizer.lr)))
72     K.set_value(model.optimizer.lr, 1e-5)
73     print("[INFO] new learning rate: {}".format(
74         K.get_value(model.optimizer.lr)))
```

Our `callbacks` list will consist of checkpointing ResNet weights to disk every five epochs, followed by plotting the training history:

```
76 # construct the set of callbacks
77 callbacks = [
78     EpochCheckpoint(args["checkpoints"], every=5,
79         startAt=args["start_epoch"]),
    TrainingMonitor(config.FIG_PATH, jsonPath=config.JSON_PATH,
        startAt=args["start_epoch"])]
```

Finally, we'll kick off the training process and train our network using mini-batches of size 64:

```
# train the network
model.fit_generator(
    trainGen.generator(),
    steps_per_epoch=trainGen.numImages // 64,
    validation_data=valGen.generator(),
    validation_steps=valGen.numImages // 64,
    epochs=50,
    max_queue_size=10,
    callbacks=callbacks, verbose=1)

# close the databases
trainGen.close()
valGen.close()
```

The exact number of epochs we'll need to train ResNet is unknown at this point, so we'll set `epochs` to a large number and adjust as needed.

ResNet on Tiny ImageNet: Experiment #1

To start training, I executed the following command:

```
$ python train.py --checkpoints output/checkpoints
```

After monitoring training for the first 25 epochs, it became clear that training loss was starting to stagnate a bit (Figure 12.9, *top-left*). To combat this stagnation, I stopped training, reduced my learning rate from $1e-1$ to $1e-2$, and resumed training:

```
$ python train.py --checkpoints output/checkpoints \
    --model output/checkpoints/epoch_25.hdf5 --start-epoch 25
```

Training continued from epochs 25-35 at this lower learning rate. We can immediately see the benefit of lowering the learning rate by an order of magnitude – loss drops dramatically, and accuracy enjoys a nice bump. (Figure 12.9, *top-right*).

However, after this initial bump, the training loss continued to drop at a much faster rate than the validation loss. I once again stopped training at epoch 35, lowered the learning rate from $1e-2$ to $1e-3$, and resumed training:

```
$ python train.py --checkpoints output/checkpoints \
    --model output/checkpoints/epoch_35.hdf5 --start-epoch 35
```

I only allowed ResNet to train for another 5 epochs as I started to notice clear signs of overfitting (Figure 12.9, *bottom*). Loss dips slightly upon the $1e-3$ change, then starts to *increase*, all the while training loss decreases at a faster rate – this is a telltale sign of overfitting. I stopped training altogether at this point, and noted the validation accuracy to be 53.14%.

To determine the accuracy on the testing set, I executed the following command (keeping in mind that the `rank_accuracy.py` script is *identical* to the one from Chapter 11 on GoogLeNet:

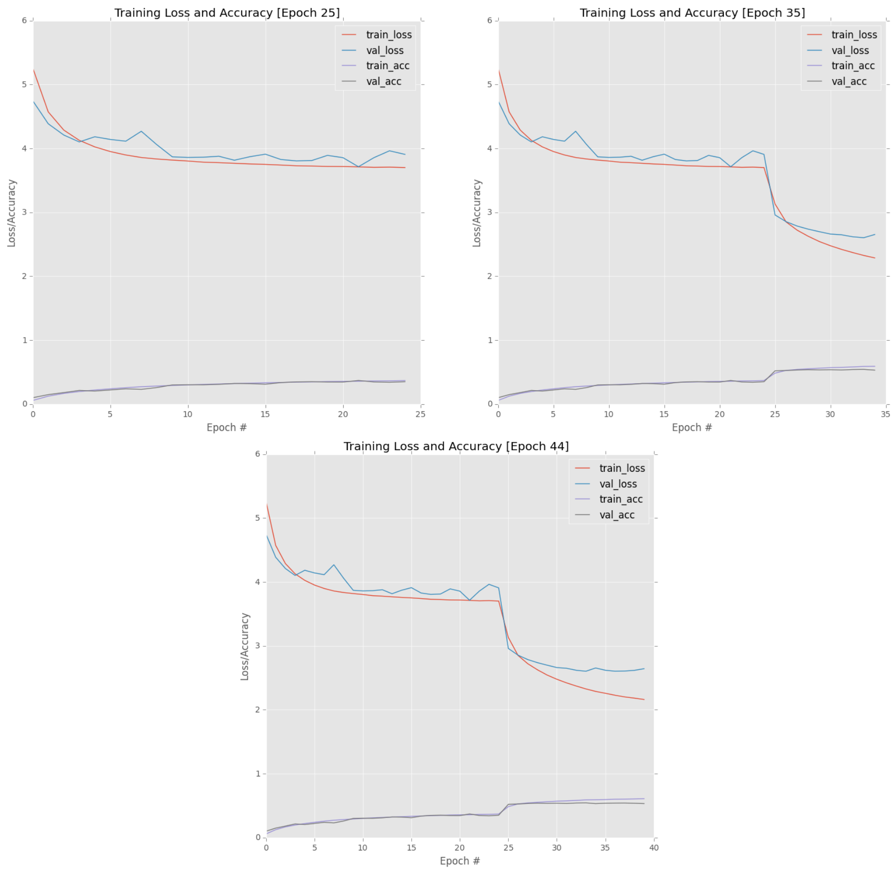


Figure 12.9: **Top-left:** First 25 epochs when training ResNet on Tiny ImageNet in Experiment #1. **Top-right:** Next 10 epochs. **Bottom:** Final 5 epochs. Notice the telltale sign of overfitting as validation loss starts to increase during the final epochs.

```
$ python rank_accuracy.py
[INFO] loading model...
[INFO] predicting on test data...
[INFO] rank-1: 53.10%
[INFO] rank-5: 75.43%
```

As the output demonstrates, we are obtaining 53.10% rank-1 accuracy on the testing set. This first experiment was not a bad one as we are already closing in on the GoogLeNet + Tiny ImageNet accuracy. Given the success of applying learning rate decay to Tiny ImageNet from the previous chapter, I immediately decided to apply the same process to ResNet.

12.5.3 Training ResNet on Tiny ImageNet with Learning Rate Decay

To train ResNet using learning rate decay, open up a new file, name it `train_decay.py`, and insert the following code:

```
1  # set the matplotlib backend so figures can be saved in the background
2  import matplotlib
3  matplotlib.use("Agg")
4
5  # import the necessary packages
6  from config import tiny_imagenet_config as config
7  from pyimagesearch.preprocessing import ImageToArrayPreprocessor
8  from pyimagesearch.preprocessing import SimplePreprocessor
9  from pyimagesearch.preprocessing import MeanPreprocessor
10 from pyimagesearch.callbacks import TrainingMonitor
11 from pyimagesearch.io import HDF5DatasetGenerator
12 from pyimagesearch.nn.conv import ResNet
13 from tensorflow.keras.preprocessing.image import ImageDataGenerator
14 from tensorflow.keras.callbacks import LearningRateScheduler
15 from tensorflow.keras.optimizers import SGD
16 import argparse
17 import json
18 import sys
19 import os
20
21 # set a high recursion limit so Theano doesn't complain
22 sys.setrecursionlimit(5000)
```

Lines 6-19 import our required Python packages. We'll need to import the `LearningRateScheduler` class on **Line 14** so we can apply a learning rate schedule to the training process. The system recursion limit is then set on **Line 22** just in case you are using the Theano backend.

Next, we define the actual function responsible for applying the learning rate decay:

```
24 # define the total number of epochs to train for along with the
25 # initial learning rate
26 NUM_EPOCHS = 75
27 INIT_LR = 1e-1
28
29 def poly_decay(epoch):
30     # initialize the maximum number of epochs, base learning rate,
31     # and power of the polynomial
32     maxEpochs = NUM_EPOCHS
33     baseLR = INIT_LR
34     power = 1.0
35
36     # compute the new learning rate based on polynomial decay
37     alpha = baseLR * (1 - (epoch / float(maxEpochs))) ** power
38
39     # return the new learning rate
40     return alpha
```

Here we indicate that we'll train for a maximum of 75 epochs (**Line 26**) with a base learning rate of $1e-1$ (**Line 27**). The `poly_decay` function is defined on **Line 29** which accepts a single parameter, the current `epoch` number. We set `power=1.0` on **Line 34** to turn the polynomial decay into a *linear* one (see Chapter 11 for more details). The new learning rate (based on the current epoch) is computed on **Line 37** and then returned to the calling function on **Line 40**.

Let's move on to the command line arguments:

```python
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-m", "--model", required=True,
     help="path to output model")
ap.add_argument("-o", "--output", required=True,
     help="path to output directory (logs, plots, etc.)")
args = vars(ap.parse_args())
```

Here we simply need to provide a path to our output serialized `--model` after training is complete along with an `--output` path to store any plots/logs.

We can now initialize our data augmentation class and load the RGB means from disk:

```python
# construct the training image generator for data augmentation
aug = ImageDataGenerator(rotation_range=18, zoom_range=0.15,
     width_shift_range=0.2, height_shift_range=0.2, shear_range=0.15,
     horizontal_flip=True, fill_mode="nearest")

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

As well as create our training and validation `HDF5DatasetGenerators`:

```python
# initialize the image preprocessors
sp = SimplePreprocessor(64, 64)
mp = MeanPreprocessor(means["R"], means["G"], means["B"])
iap = ImageToArrayPreprocessor()

# initialize the training and validation dataset generators
trainGen = HDF5DatasetGenerator(config.TRAIN_HDF5, 64, aug=aug,
     preprocessors=[sp, mp, iap], classes=config.NUM_CLASSES)
valGen = HDF5DatasetGenerator(config.VAL_HDF5, 64,
     preprocessors=[sp, mp, iap], classes=config.NUM_CLASSES)
```

Our `callbacks` list will consist of a `TrainingMonitor` and a `LearningRateScheduler`:

```python
# construct the set of callbacks
figPath = os.path.sep.join([args["output"], "{}.png".format(
     os.getpid())])
jsonPath = os.path.sep.join([args["output"], "{}.json".format(
     os.getpid())])
callbacks = [TrainingMonitor(figPath, jsonPath=jsonPath),
     LearningRateScheduler(poly_decay)]
```

Below we instantiate our `ResNet` architecture and SGD optimizer using the same parameters as our first experiment:

```python
# initialize the optimizer and model (ResNet-56)
print("[INFO] compiling model...")
model = ResNet.build(64, 64, 3, config.NUM_CLASSES, (3, 4, 6),
     (64, 128, 256, 512), reg=0.0005, dataset="tiny_imagenet")
```

```python
opt = SGD(lr=INIT_LR, momentum=0.9)
model.compile(loss="categorical_crossentropy", optimizer=opt,
     metrics=["accuracy"])
```

Finally, we can train our network in mini-batches of 64:

```python
# train the network
print("[INFO] training network...")
model.fit_generator(
     trainGen.generator(),
     steps_per_epoch=trainGen.numImages // 64,
     validation_data=valGen.generator(),
     validation_steps=valGen.numImages // 64,
     epochs=NUM_EPOCHS,
     max_queue_size=10,
     callbacks=callbacks, verbose=1)
```

The number of epochs we are going to train for is controlled by NUM_EPOCHS defined earlier in this script. We'll be linearly decreasing our learning rate from 1e-1 down to zero over the course of NUM_EPOCHS. And finally serialize the model to disk once training is complete:

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

# close the databases
trainGen.close()
valGen.close()
```

ResNet on Tiny ImageNet: Experiment #2

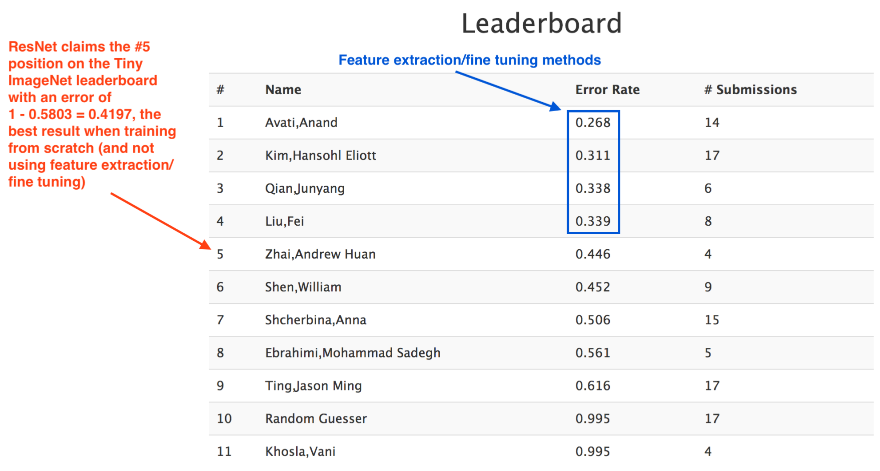
In this experiment, I trained the ResNet architecture detailed above on the Tiny ImageNet dataset using the SGD optimizer, a base learning rate of $1e-1$, and a momentum term of 0.9. The learning rate was decayed linearly over the course of 75 epochs. To perform this experiment, I executed the following command:

```
$ python train_decay.py --model output/resnet_tinyimagenet_decay.hdf5 \
     --output output
```

The resulting learning plot can be found in Figure 12.11. Here we can see the dramatic effect that learning rate decay can have on a training process. While both training loss and validation loss diverge from each other, it's not until epoch 60 where the gap widens past previous epochs. Furthermore, our validation loss continues to *decrease* as well. At the end of the 75th epoch, I obtained 58.32% rank-1 accuracy.

I then evaluated the network on the testing set using the following command:

```
$ python rank_accuracy.py
[INFO] loading model...
[INFO] predicting on test data...
[INFO] rank-1: 58.03%
[INFO] rank-5: 80.46%
```


ResNet claims the #5 position on the Tiny ImageNet leaderboard with an error of 1 - 0.5803 = 0.4197, the best result when training from scratch (and not using feature extraction/ fine tuning)
Feature extraction/fine tuning methods

Leaderboard

#	Name	Error Rate	# Submissions
1	Avati,Anand	0.268	14
2	Kim,Hansohl Eliott	0.311	17
3	Qian,Junyang	0.338	6
4	Liu,Fei	0.339	8
5	Zhai,Andrew Huan	0.446	4
6	Shen,William	0.452	9
7	Shcherbina,Anna	0.506	15
8	Ebrahimi,Mohammad Sadegh	0.561	5
9	Ting,Jason Ming	0.616	17
10	Random Guesser	0.995	17
11	Khosla,Vani	0.995	4

Figure 12.10: Using ResNet we are able to reach the #5 position on the Tiny ImageNet leaderboard, beating out all other approaches that attempted to train a network from scratch. All results with lower error than our approach applied fine-tuning/feature extraction.

As the results demonstrated, we have reached **58.03%** rank-1 and **80.46%** rank-5 accuracy on the testing set, a *substantial* improvement over our previous chapter on GoogLeNet. This result leads to a test error of $1 - 0.5803 = 0.4197$, which *easily* takes position #5 on the Tiny ImageNet leaderboard (Figure 12.10).

This is the *best accuracy* obtained on the Tiny ImageNet dataset that *does not* perform some form of transfer learning or fine-tuning. Given that the accuracies for positions 1-4 were obtained from fine-tuning a network that was *already* trained on the fully ImageNet dataset, it's hard to compare a fine-tuned network to one that was trained entirely from scratch. If we (fairly) compare our network that was trained from scratch to the other networks trained from scratch on the leaderboard, we can see that our ResNet has obtained the highest accuracy amongst the group.

12.6 Summary

In this chapter, we discussed the ResNet architecture in detail, including the residual module micro-architecture. The original residual module proposed by He et al. in their 2015 paper, *Deep Residual Learning for Image Recognition* [24] has gone through many revisions. Originally the module consisted of two `CONV` layers and an identity mapping "shortcut". In the same paper, it was found that adding the "bottleneck" sequence of 1×1, 3×3, and 1×1 `CONV` layers improved accuracy.

Then, in their 2016 study, *Identity Mappings in Deep Residual Networks* [33], the *pre-activation* residual module was introduced. We call this update "pre-activation" because we apply the activation and batch normalization *before* the convolution, going against the "conventional wisdom" when building Convolutional Neural Networks.

From there, we implemented the ResNet architecture using both bottleneck and pre-activation using the Keras framework. This implementation was then used to train ResNet on both the CIFAR-10 and Tiny ImageNet datasets. In CIFAR-10, we were able to replicate the results of He et al., obtaining **93.58%** accuracy. Then, on Tiny ImageNet, reached **58.03%** accuracy, the highest accuracy thus far from a network *trained from scratch* on Stanford's cs231n Tiny ImageNet

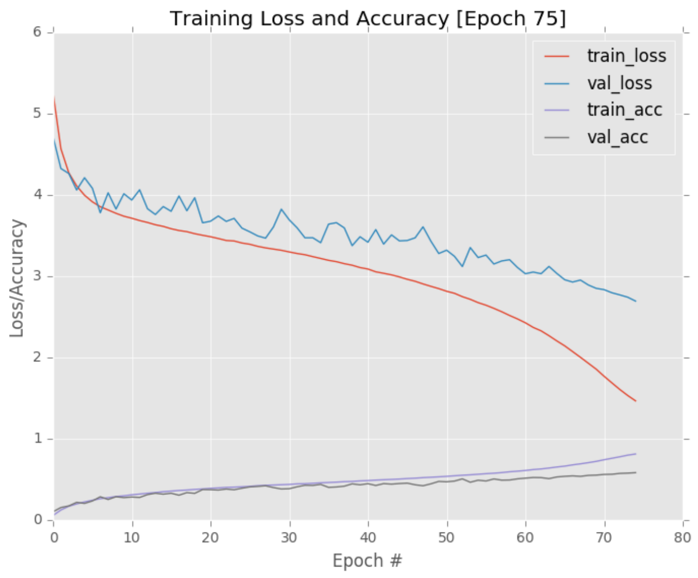


Figure 12.11: Applying learning rate to ResNet when trained on Tiny ImageNet leads to 58.32% validation and 58.03% testing accuracy – significantly higher than our previous experiment using `ctrl + c` training.

challenge.

Later in the *ImageNet Bundle*, we'll investigate ResNet and train it on the *complete* ImageNet dataset, once again replicating the work of He et al.

21. Mask R-CNN and Cancer Detection

So far in this book we've explored two primary forms of image understanding: *classification* and *detection*. When performing image classification our goal is to predict a single label that characterizes the contents of the entire image.

Object detection on the other hand seeks to (1) find *all* objects in an image, (2) label them, and (3), compute their bounding box (x, y)-coordinates, enabling us to not only determine *what* is in an image but also *where* in the image an object resides.

Object detection is clearly a harder task than image classification. But a task even *harder* than object detection is *segmentation* and more specifically, *instance segmentation.*

Instance segmentation takes object detection a step further — instead of predicting a bounding box for each object in an image, we now see to *predict a mask* for each object, giving us a pixel-wise segmentation of the object rather than a coarse, perhaps even unreliable bounding box. Instance segmentation algorithms power much of the "magic" we see in computer vision, including self-driving cars, robotics, and more.

In this chapter we will start with a brief discussion of object detection, semantic segmentation, and instance segmentation, including the differences between them.

From there we'll review the Mask R-CNN architecture [71] which can be used for instance segmentation. We'll also explore Keras Mask R-CNN [72], an implementation of the Mask R-CNN framework that we'll be using in this chapter along with the following one.

Once we understand how the Mask R-CNN framework is used for instance segmentation I'll be showing you how to train your own instance segmentation networks for cancer detection, specifically skin lesion analysis for melanoma detection.

21.1 Object Detection vs. Instance Segmentation

Deep learning has helped facilitate unprecedented accuracy in computer vision, including image classification, object detection, and now even *segmentation.* So, what is the difference between these types of algorithms?

As we learned from earlier in the *Practitioner Bundle* and *ImageNet Bundle* of this book we know that object detection is the process of (1) detecting objects in an image followed by

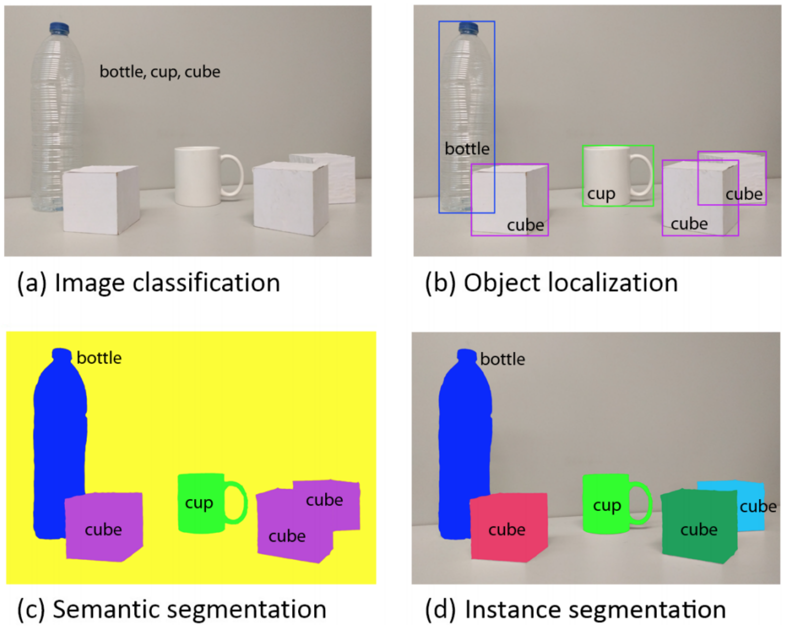


Figure 21.1: **Top-left:** When performing *image classification* we predict a set of labels for an input image. **Top-right:** *Object detection* allows us to predict the bounding box (x, y)-coordinates for *each* object in an image along with the class label. *Bottom-left: Semantic segmentation* associates *every* pixel in an image with a label. **Bottom-right:** *Instance segmentation* computes a pixel-wise mask for *every* object in the image, even if objects are of the same class label. Figure from Garcia-Garcia et al. [73]

(2) predicting their corresponding bounding box coordinates and class labels. An example of performing object detection can be seen in Figure 21.1 *(top-right)*.

Traditional segmentation on the other hand involves partitioning an image into parts. Popular traditional segmentation algorithms include Normalized Cuts [74], Graph Cuts [75], Grab Cuts [76], and superpixels [77, 78, 79, 80]; *however*, none of these algorithms have any actual *understanding* of what each of the resulting segmented image parts represent.

Semantic segmentation (Figure 21.1, *bottom-left*) and instance segmentation (Figure 21.1, *bottom-right*) algorithms seek to introduce *understanding* into their output parts.

Specifically, semantic segmentation algorithms attempt to:

- *Partition* the image into meaningful parts
- While at the same time, *associating every pixel* in an input image with a class label (i.e., person, road, car, bus, etc.).

Semantic segmentation algorithms have many use cases but are most popularly used in self-driving cars where we seek to understand the *entire scene* that a car camera is looking at. Semantic segmentation algorithms may also require that each of the objects, regardless of class, be *uniquely labeled* as well. This type of segmentation is called a *panoptic segmentation*.

It's important to note that panoptic segmentation is an *extremely* challenging problem since we need to ensure that not only *every* pixel in an input image is assigned to a label, but also each object is *uniquely labeled* and segmented as well.

Instance segmentation on the other hand is a some-what easier problem (compared to panoptic segmentation). Instance segmentation can be thought of a hybrid between both object detection and semantic segmentation.

While object detection produces a *bounding box*, instance segmentation produces a pixel-wise *mask* for each individual object object. Instance segmentation does not require every pixel in an image to be associated with a label, however.

Figure 21.1 provides visualizations for image classification *(top-left)*, object detection *(top-right)*, semantic segmentation *(bottom-left)*, and instance segmentation *(bottom-right)*. Since instance segmentation tends to be a more tractable, realistic use case for most problems, and not to mention, there is more labeled data for such a task, we'll be focusing on performing instance segmentation with the Mask R-CNN framework in this chapter along with the following chapter.

21.1.1 What is Mask R-CNN?

The Mask R-CNN algorithm was introduced by He et al. in their 2017 paper, *Mask R-CNN*. Mask R-CNN builds on the previous object detection work of R-CNN, Fast R-CNN, and Faster R-CNN by Girshick et al. [39, 42, 48]

R The Faster R-CNN family of object detectors is covered in detail inside Section 14.2 of Chapter 14 of the *ImageNet Bundle* — please refer to this chapter if you need a refresher on how the Faster R-CNN algorithm works as I'll be assuming you have read the chapter before continuing your study of Mask R-CNN.

As we know from our previous study of Faster R-CNN in the *ImageNet Bundle*, the Faster R-CNN architecture takes the output of the ROI Pooling module and passes it through two FC layers (each 4096-d), similar to the final layers of classification networks of ImageNet.

The output of these `FC` layers feed into the final two `FC` layers of the network. One `FC` layer is $N+1$-d, a node for each of the class labels plus an additional label for the background. The second `FC` layer is $4\times N$ which represents the deltas for final predicted bounding box.

Thus, we can think of a Faster R-CNN as producing two sets of values:

- The class label probabilities
- The corresponding bounding box predictions

A visualization of the final Faster R-CNN component of the architecture can be seen in Figure 21.2 *(top)*.

To turn our Faster R-CNN into a Mask R-CNN, He et al. proposed adding an *additional branch* to the network. This additional branch is responsible for predicting the actual *mask* of an object/class. A visualization of the updated architecture can be see in Figure 21.2 *(bottom)*.

Notice how a new parallel branch has been introduced in the network architecture. The masking branch splits off from the ROI Align module (an improvement to the Girshick et al. ROI Pool module) *prior* to our `FC` layers and then consists of two `CONV` layers responsible for creating the mask predictions themselves.

In practice, Mask R-CNN introduces only a small overhead to the Faster R-CNN framework and can still run at approximately 5 FPS with a. ResNet-101 backend on a GPU [71]. Furthermore, just like Faster R-CNN and SSD can leverage different architectures such as ResNet, VGG, SqueezeNet, MobileNet, etc. as their backend/backbone, making it possible to decrease mobile size and potentially increase FPS throughput as well.

Understanding Mask R-CNN Output Volume Size

Before we get too far into this chapter I believe it's worth having a discussion on the output volume dimensions of a Mask R-CNN.

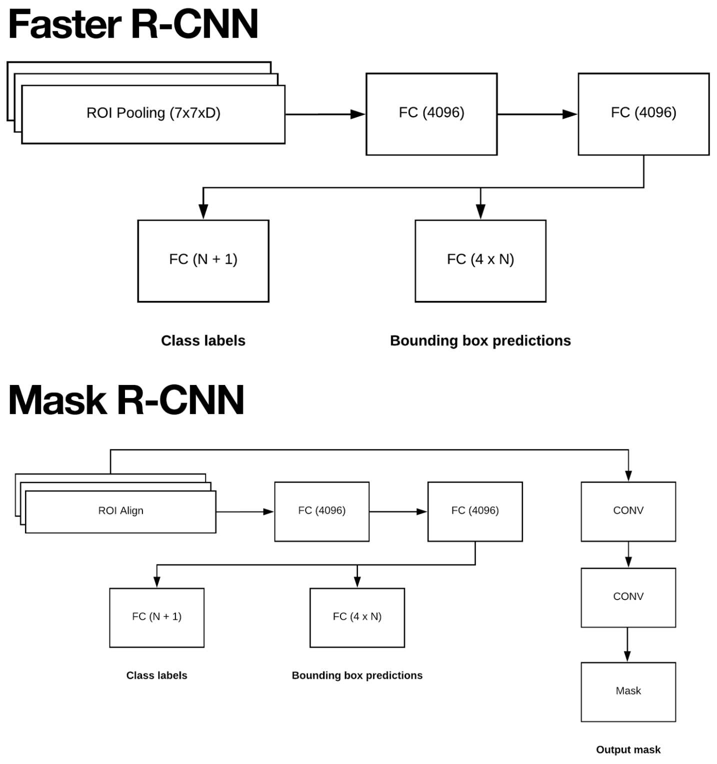


Figure 21.2: **Top:** The original output layers of the Faster R-CNN framework [42]. One `FC` layer contains our bounding box offset predictions while the second `FC` layer contains the corresponding class labels for each bounding box. **Bottom:** The Mask R-CNN work by [71] replaces the ROI Pooling module with a more accurate ROI Align module. The output of the ROI module is then fed into two `CONV` layers. The output of these `CONV` layers is the mask itself.

As we know from Chapter 14 of the *ImageNet Bundle*, the Faster R-CNN/Mask R-CNN leverages a Region Proposal Network (RPN) to generate regions of an image that *potentially* contain an object.

Each of these regions is ranked based on their "objectness score" (i.e., how likely it is a given region could potentially contain an object) and then the top N most confident objectness regions are kept.

In the original Faster R-CNN publication, Girshick et al. set $N = 2,000$, but in practice we can get away with a much smaller N, such as $N = 10, 100, 200, 300$ and still obtain good results. He et al. set $N = 300$ in their publication which is the value we'll use here as well.

Each of the 300 most ROIs then go through the three parallel branches of the network:

- Label prediction
- Bounding box prediction
- Mask prediction

Figure 21.2 from earlier in this chapter provides a visualization of each of these branches.

During prediction, each of the 300 ROIs go through non-maxima suppression [81] and the top 100 detection boxes are kept, resulting in a 4D tensor of $100 \times L \times 15 \times 15$ where L is the number

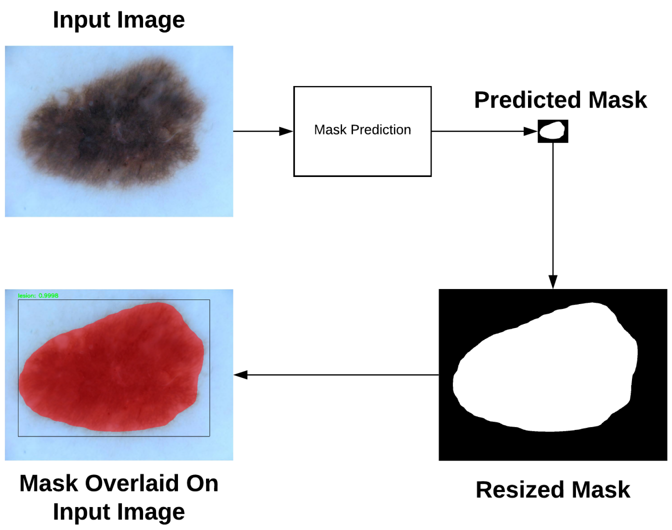


Figure 21.3: We start with our input image and then feed it through our Mask R-CNN network to obtain our mask prediction. The predicted mask is only 15×15 pixels so we apply nearest neighbor interpolation to resize it back to the original image dimensions. The resized mask can then be overlaid on the original input image.

of class labels in the dataset and 15×15 is the size of each of the L masks.

For example, if we're using the COCO dataset, which has $L = 90$ classes, then the resulting volume size from the mask module of the Mask R-CNN will be $100 \times 90 \times 15 \times 15$.

After performing a forward pass we can loop over each of the detected bounding boxes, find the class label index with the largest corresponding probability, and then use the index lookup the 15×15 mask in the $100 \times 90 \times 15 \times 15$ volume.

Written in pseudocode our lookup may look like: `instanceMask = masks[i, classID]`, where i is the index of the current detection (i.e., the first dimension of the output volume) and `classID` is the index of the class label with the largest corresponding probability.

After performing the lookup we now have `instanceMask`, a 15×15 mask that represents the pixel-wise segmentation of the i-th detected object.

We can then scale the mask back to the original dimensions of the image *making sure to use nearest neighbor interpolation*, allowing us to obtain the pixel-wise mask for the original image.

An example of extracting a 15×15 mask, resizing to the original image dimensions, and then overlaying the mask on the detected object can be seen in Figure 21.3.

So, why nearest neighbor interpolation? A discussion of interpolation algorithms is well outside the scope of this book and, is not to mention, a field of study itself; however, the gist is that nearest neighbor interpolation will preserve all original values of the image during resizing without introducing new values.

While nearest neighbor interpolation is less appeasing to the human eye, such as bilinear or bicubic interpolation, for example, the benefit here is that we do not introduce "new" object/mask assignments when resizing the image. Make sure when resizing your own masks that you are performing nearest neighbor interpolation as well.

For a more detailed review of interpolation algorithms, including visual examples, please refer to the following excellent article on MathWorks: http://pyimg.co/k8r5r.

Additionally, if you would like more information on Mask R-CNN, including more visual examples on volume size and how we use to derive the resulting mask for an object, please refer to the *Mask R-CNN with OpenCV* blog post on the PyImageSearch site: http://pyimg.co/4lwab.

I would *highly encourage* you to read the blog post as it will give you more exposure to the Mask R-CNN architecture (namely how to process output predictions), making it easier for you to grasp, comprehend, and retain the information in the rest of this chapter.

21.2 Installing Keras Mask R-CNN

To actually train our own custom Mask R-CNN models, we'll be using the Matterport Keras + Mask R-CNN implementation [72]. The Matterport implementation is arguably one of the most used and stable Keras implementations of Mask R-CNN.

Furthermore, the framework is fairly intuitive, but there are a few aspects that can and will trip you up as you get started with the library. I learned first-hand with this library and will be sharing my experience along the way to ensure you do not run into the same problems I did.

Just like RetinaNet, the Keras Mask R-CNN implementation is still considered "under development" with no official releases. Since libraries and packages under development can and will change frequently, I've decided to move the installing and configuration section to the companion website where I can more easily update the install instructions as they change.

To install the Keras Mask R-CNN library on your machine, please refer to this link: http://pyimg.co/uib16.

And if you do not already have an account on the companion website, please refer to the first few pages of this book where we'll find a link to create your companion website account.

21.3 The ISIC Skin Lesion Dataset

The dataset we'll be using in this case study is the International Skin Imaging Collaboration (ISIC) 2018 Skin Lesions dataset [82, 83]. This particular dataset enables computer vision researchers, developers, and engineers to contribute to the study and early detection of cancer, in particular melanoma.

Melanoma is a major public health concern with over 5,000,000 newly diagnosed cases per year in the United States alone. Melanoma is also the deadliest form of skin cancer with 350,000 reported cases and 60,000 deaths in 2015.

The good news is that when caught early, melanoma survival rates are in excess of 95%. Therefore, early detection of the cancer is key.

In mid-2018, ISIC released a three part imaging challenge, including:

- Lesion segmentation
- Lesion attribute detection
- Disease classification

The goal of lesion segmentation is to automatically segment the boundaries of potential cancerous regions from dermoscopic images — this challenge lends itself *perfectly* to instance segmentation and is the one we will be examining in this case study.

You can find an example of skin lesion boundary detection in Figure 21.4. The *top* row contains examples of skin lesion images while the *bottom* row provides their corresponding masks. Our goal will be to correctly predict those pixel-wise masks from the input image.

The second task is more challenging, involving automatically predicting and localizing various attributes of the lesion, such as pigment networks, streaks, milia-like cysts, and more.

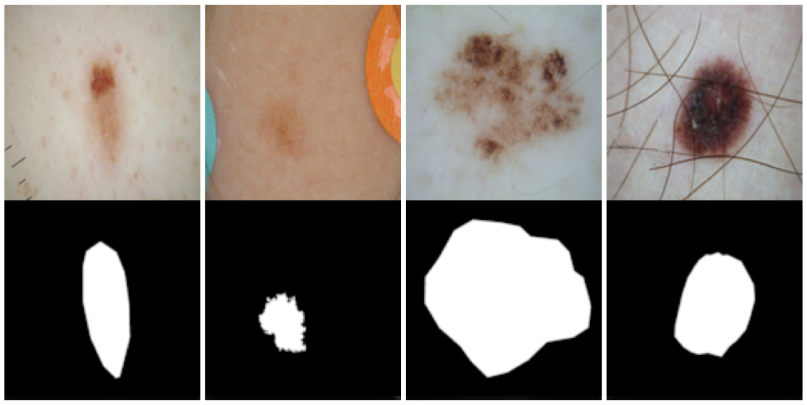


Figure 21.4: The dataset we'll be using for this chapter is from the International Skin Imaging Collaboration (ISIC) 2018 Skin Lesions challenge [82, 83]. We are provided with the skin lesion images *(top)* along with their corresponding masks (*bottom*). Our goal is to train a Mask R-CNN to accurately predict such masks for new skin lesion inputs. These masks can aid a computer vision-based system in predicting and classifying melanoma, a type of skin cancer.

The final task is simple image classification — given an input image the goal is to classify the lesion into seven distinct categories such as Melanoma, Basal cell carcinoma, Melanocytic nevus, and others.

You can learn more about the ISIC 2018 dataset, including more information regarding each of the three challenges, from their official challenge page (http://pyimg.co/u3nuv). In the context of the rest of this chapter, we'll be focusing primarily on the first task — skin lesion boundary detection.

21.3.1 How Do I Download the ISIC 2018 Dataset?

The ISIC 2018 dataset is freely available for you to use and study from for non-commercial purposes. To access the dataset you first need to create an account on the ISIC 2018 website: http://pyimg.co/h7uxv.

After registering you'll be able to click the *"Download training dataset"* and *"Download ground truth data"* buttons to download .zip archives of the training data and corresponding ground-truth masks (Figure 21.5).

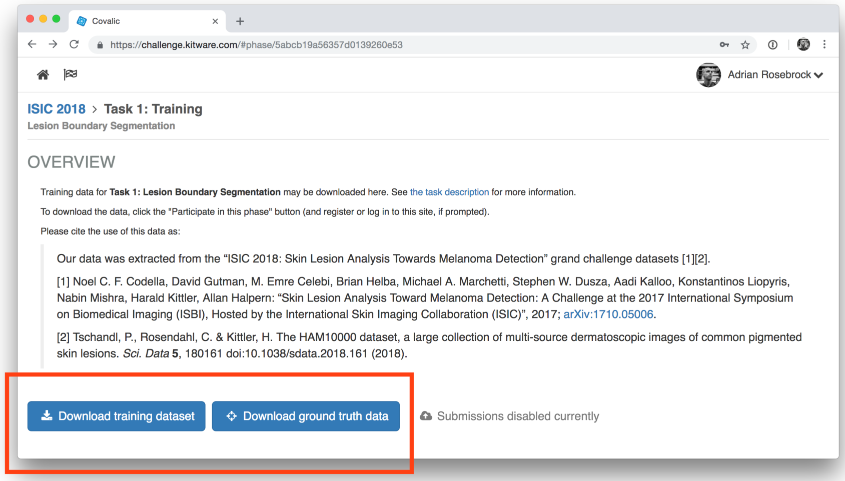
21.3.2 Skin Lesion Boundary Detection Challenge

The skin lesion boundary segmentation dataset consists of 2,594 images in JPEG format (11GB) and 2,594 corresponding ground-truth masks in PNG format (27MB).

Why PNG? PNG is a *lossless* image file format, implying that the data compression algorithm used for PNG images allows the original data to be 100% *perfectly* reconstructed from compressed data.

Lossy file formats on the other hand, such as JPEG, can reduce the actual size of our images, and most of the time without our eyes being able to tell the difference.

The reason we use lossless image file formats for the mask is to (1) ensure the pixel-wise segmentation of the mask is perfectly retained and (2) that any pixel values themselves are perfectly retained. If we used a lossless file format those pixel values may change slightly, perhaps rendering


Covalic
https://challenge.kitware.com/#phase/5abcb19a56357d0139260e53
Adrian Rosebrock

ISIC 2018 > Task 1: Training
Lesion Boundary Segmentation

OVERVIEW

Training data for **Task 1: Lesion Boundary Segmentation** may be downloaded here. See the task description for more information.

To download the data, click the "Participate in this phase" button (and register or log in to this site, if prompted).

Please cite the use of this data as:

> Our data was extracted from the “ISIC 2018: Skin Lesion Analysis Towards Melanoma Detection” grand challenge datasets [1][2].
>
> [1] Noel C. F. Codella, David Gutman, M. Emre Celebi, Brian Helba, Michael A. Marchetti, Stephen W. Dusza, Aadi Kalloo, Konstantinos Liopyris, Nabin Mishra, Harald Kittler, Allan Halpern: “Skin Lesion Analysis Toward Melanoma Detection: A Challenge at the 2017 International Symposium on Biomedical Imaging (ISBI), Hosted by the International Skin Imaging Collaboration (ISIC)”, 2017; arXiv:1710.05006.
>
> [2] Tschandl, P., Rosendahl, C. & Kittler, H. The HAM10000 dataset, a large collection of multi-source dermatoscopic images of common pigmented skin lesions. *Sci. Data* **5**, 180161 doi:10.1038/sdata.2018.161 (2018).

Download training dataset
Download ground truth data
Submissions disabled currently

Figure 21.5: After registering and logging in, you will be able to download bout the training images and corresponding ground-truth masks.

our masks unusable.

All masks are encoded as single-channel (grayscale) images. Each pixel in the mask has only one of two values:

1. 0: Areas *outside* the lesion (i.e., background)
2. 255: Areas *inside* the lesion (i.e., foreground)

After unzipping your two training data you should have the following directory structure:

```
$ ls -l
ISIC2018_Task1-2_Training_Input
ISIC2018_Task1-2_Training_Input.zip
ISIC2018_Task1_Training_GroundTruth
ISIC2018_Task1_Training_GroundTruth.zip
```

Notice how the training images and ground-truth directories are clearly named as such.

Let’s take a look at the filenames of our training images:

```
$ ls -l ISIC2018_Task1-2_Training_Input/*.jpg | head -n 5
ISIC2018_Task1-2_Training_Input/ISIC_0000000.jpg
ISIC2018_Task1-2_Training_Input/ISIC_0000001.jpg
ISIC2018_Task1-2_Training_Input/ISIC_0000003.jpg
ISIC2018_Task1-2_Training_Input/ISIC_0000004.jpg
ISIC2018_Task1-2_Training_Input/ISIC_0000006.jpg
```

Note how all input images have the format `ISIC_<image_id>.jpg` where `image_id` is a 7-digit unique identifier. We can use this 7-digit ID to associate our input images to their corresponding masks.

Our mask images have a similar filename structure:

```
$ ls -l ISIC2018_Task1_Training_GroundTruth/*.png | head -n 5
ISIC2018_Task1_Training_GroundTruth/ISIC_0000000_segmentation.png
ISIC2018_Task1_Training_GroundTruth/ISIC_0000001_segmentation.png
ISIC2018_Task1_Training_GroundTruth/ISIC_0000003_segmentation.png
ISIC2018_Task1_Training_GroundTruth/ISIC_0000004_segmentation.png
ISIC2018_Task1_Training_GroundTruth/ISIC_0000006_segmentation.png
```

Notice the only difference between the input images and their corresponding masks is that the mask images (1) include `_segmentation` and the end of their filename and (2) are a PNG filetype.

Given the similarities between the filenames it will be uncomplicated and straightforward for us to derive the mask filename from an input image filename and vice versa.

21.4 Training Your Mask R-CNN

Now that we've explored our skin lesion dataset, let's start exploring how we can train a Mask R-CNN to predict the boundaries of the lesion areas.

21.4.1 Project Structure

Before we can start training our Mask R-CNN, we first need to examine our project structure:

```
|--- mask_rcnn
|    |--- isic2018/
|    |    |--- ISIC2018_Task1-2_Training_Input
|    |    |--- ISIC2018_Task1_Training_GroundTruth
|    |--- lesions.py
|    |--- mask_rcnn_coco.h5
|    |--- mrcnn/
```

The `isic2018` directory contains the ISIC 2018 dataset itself. You can either store the unzipped directories from Section 21.3.2 in this directory or you can do what I did and create a sym-link to the dataset via the `ln` command — either will achieve the desired result.

The `lesions.py` file contains all code that we will need to train our Mask R-CNN. We will be implementing the code in this file in the next section.

The `mask_rcnn_coco.h5` file is a Mask R-CNN with ResNet backbone pre-trained on the COCO dataset [56]. We will be fine-tuning this model for our own segmentation tasks.

In nearly all situations it is wise to start with a *pre-trained* model and the *fine-tune* it — training from scratch not only takes longer, but is more tedious, typically requiring additional training data as well. I have included the `mask_rcnn_coco.h5` file in the downloads associated with this book but you can also download the file here: http://pyimg.co/r5992.

Finally, the `mrcnn` directory contains the Matterport Keras + Mask R-CNN implementation.

If you followed by Keras + Mask R-CNN install sections from Section 21.2 then you should have cloned the Matterport Mask R-CNN GitHub repository into your home directory. From there, you can either (1) copy the `Mask-RCNN/mrcnn` directory into this project or (2) do what I do and create a sym-link:

```
$ ln -s ~/Mask_RCNN/mrcnn mrcnn
```

Now that we've reviewed our project structure, let's go ahead and implementing the training script!

21.4.2 Implementing the Mask R-CNN

To get started, open up the `lesions.py` file and insert the following code:

```
1  # import the necessary packages
2  from imgaug import augmenters as iaa
3  from mrcnn.config import Config
4  from mrcnn import model as modellib
5  from mrcnn import visualize
6  from mrcnn import utils
7  from imutils import paths
8  import numpy as np
9  import argparse
10 import imutils
11 import random
12 import cv2
13 import os
```

Lines 2-13 handle importing our required Python packages and libraries. Notice how **Line 2** imports from the `imgaug` library [84].

As we know from chapters in both the *Practitioner Bundle* and *ImageNet Bundle*, image augmentation allows us to create additional training data on the fly by applying random transformations to our input images, including rotation, translation, scaling, etc. Image augmentation is often *critical* in reducing overfitting and increasing the ability of your model to generalize.

That said, applying image augmentation to instance segmentation isn't as easy or straightforward as classification — any transformation that is performed to the input *image* also needs to be performed on the *mask* as well.

The `imgaug` library enables us to perform more powerful augmentation than what the standard Keras `ImageDataGenerator` does. For more information on `imgaug`, please refer to their official GitHub repository [84].

Lines 3-6 import our Mask R-CNN classes and functions. We'll be subclassing the `Config` class in order to derive our own configuration for training on the ISIC 2018 dataset.

The `modellib` contains the Mask R-CNN model itself. The `visualize` submodule will help us visualize the output predictions of the Mask R-CNN while the `utils` submodule contains various utilities we'll be leveraging in this script.

Next, let's derive the path to both our images and masks:

```
15 # initialize the dataset path, images path, and annotations file path
16 DATASET_PATH = os.path.abspath("isic2018")
17 IMAGES_PATH = os.path.sep.join([DATASET_PATH,
18     "ISIC2018_Task1-2_Training_Input"])
19 MASKS_PATH = os.path.sep.join([DATASET_PATH,
20     "ISIC2018_Task1_Training_GroundTruth"])
```

The `DATASET_PATH` is the root directory of the ISIC 2018 dataset. We then use the `DATASET_PATH` to derive the path to where all input skin lesion images are stored (`IMAGES_PATH`) as well as their corresponding masks (`MASKS_PATH`).

Given these paths we can then construct our training and validation split:

```
22 # initialize the amount of data to use for training
23 TRAINING_SPLIT = 0.8
```

```
24
25 # grab all image paths, then randomly select indexes for both training
26 # and validation
27 IMAGE_PATHS = sorted(list(paths.list_images(IMAGES_PATH)))
28 idxs = list(range(0, len(IMAGE_PATHS)))
29 random.seed(42)
30 random.shuffle(idxs)
31 i = int(len(idxs) * TRAINING_SPLIT)
32 trainIdxs = idxs[:i]
33 valIdxs = idxs[i:]
```

Line 23 indicates that we are using 80% of our data for training and 20% for validation.

From there, **Line 27** grabs all `IMAGE_PATHS` in the input `IMAGES_PATH` directory.

Line 28 generates an indexes (`idxs`) array with the same length as `IMAGE_PATHS`. We randomly shuffle the indexes (**Line 30**) and apply array slicing (**Lines 31-33**) to determine our `trainIdxs` and `valIdxs`. We use *indexes* rather than the raw file paths for reasons to become apparent later in this core review.

We also need to define a `CLASS_NAMES` dictionary:

```
35 # initialize the class names dictionary
36 CLASS_NAMES = {1: "lesion"}
37
38 # initialize the path to the Mask R-CNN pre-trained on COCO
39 COCO_PATH = "mask_rcnn_coco.h5"
40
41 # initialize the name of the directory where logs and output model
42 # snapshots will be stored
43 LOGS_AND_MODEL_DIR = "lesions_logs"
```

The `CLASS_NAMES` dictionary stores the unique integer ID of the class as the key and the human-readable class label as the value. For the ISIC 2018 skin lesion boundary detection challenge we have only *one* class — the lesion itself.

We also define the path to the pre-trained COCO model (`COCO_PATH`) as well as a directory where all output model snapshots and logs will be stored (`LOGS_AND_MODEL_DIR`).

The Matterport Mask R-CNN implementation attempts to make it as easy as possible for us to train our own Mask R-CNNs. To accomplish this task, they require us to subclass the `Config` class and override and/or define any configurations we may need.

Let's get started by examining our *training* configuration:

```
45 class LesionBoundaryConfig(Config):
46     # give the configuration a recognizable name
47     NAME = "lesion"
48
49     # set the number of GPUs to use training along with the number of
50     # images per GPU (which may have to be tuned depending on how
51     # much memory your GPU has)
52     GPU_COUNT = 1
53     IMAGES_PER_GPU = 1
54
55     # set the number of steps per training epoch and validation cycle
56     STEPS_PER_EPOCH = len(trainIdxs) // (IMAGES_PER_GPU * GPU_COUNT)
57     VALIDATION_STEPS = len(valIdxs) // (IMAGES_PER_GPU * GPU_COUNT)
```

```
58
59      # number of classes (+1 for the background)
60      NUM_CLASSES = len(CLASS_NAMES) + 1
```

The `LesionBoundaryConfig` class stores all relevant configurations when training our Mask R-CNN on the skin lesion dataset. To start, we need to give this dataset and experiment a `NAME` — we aptly name the dataset `lesion`.

Line 52 and 53 define the total number of GPUs we'll be using for training along with the number of images per GPU (i.e., batch size). I performed all experiments for this case study on a machine with a single Titan X GPU so I set my `GPU_COUNT` to one.

While my 12GB GPU could technically handle more than one image at a time, I decided to set `IMAGES_PER_GPU` to one as most readers will not have a GPU with as much memory. Feel free to increase this value if your GPU can handle it.

Line 56 defines our `STEPS_PER_EPOCH` which is simply the number of training images divided by the number of images per GPU and the GPU count. Similarly, we do the same for validation steps on **Line 57**.

Finally, `NUM_CLASSES` defines the total number of classes in the dataset which we set to be the length of the `CLASSES` dictionary plus one for the background class.

Just as we have a *training* configuration, we also have an *inference/prediction* configuration as well:

```
62  class LesionBoundaryInferenceConfig(LesionBoundaryConfig):
63      # set the number of GPUs and images per GPU (which may be
64      # different values than the ones used for training)
65      GPU_COUNT = 1
66      IMAGES_PER_GPU = 1
67
68      # set the minimum detection confidence (used to prune out false
69      # positive detections)
70      DETECTION_MIN_CONFIDENCE = 0.9
```

Take note that our inference configuration is *not* a subclass of the Mask R-CNN `Config` class but is rather a subclass of our `LesionBoundaryConfig` used for training. We perform this subclassing out of convenience, ensuring we do not have to redefine variables that have already been defined before.

That said, I have included **Lines 65 and 66** demonstrating how you can override any variables — here you can adjust your `GPU_COUNT` and `IMAGES_PER_GPU` as you see fit.

I'd also like to callout **Line 70** where we define the `DETECTION_MIN_CONFIDENCE` which controls the minimum probability required for our Mask R-CNN to mark the given region as a "detection".

Here we set the minimum required confidence to 90% to help prune out false-positive detections. A value of 90% works well for the skin lesions dataset but you may way to adjust this value for your own projects.

R When you train a Mask R-CNN on your own custom dataset I actually recommend setting `DETECTION_MIN_CONFIDENCE` to a small value, such as `0.5`. The primary reason is that you want to ensure that your network is actually learning *something*. If your minimum probability is too high then you may be unable to visually validate your network is learning. Start with a low detection confidence, validate your network is learning, and then increase it.

Our next class is responsible for actually managing the lesion dataset, including loading both images and their corresponding masks from disk:

```
71 class LesionBoundaryDataset(utils.Dataset):
72     def __init__(self, imagePaths, classNames, width=1024):
73         # call the parent constructor
74         super().__init__(self)
75 
76         # store the image paths and class names along with the width
77         # we'll resize images to
78         self.imagePaths = imagePaths
79         self.classNames = classNames
80         self.width = width
```

Here we define our `LesionBoundaryDataset` which is a subclass of the `Dataset` class in the `mrcnn` module. Defining a dataset class is a *requirement* when working with the Matterport implementation of the Mask R-CNN. The dataset class defines the logic to load an image and load a mask.

Our constructor on **Line 73** requires two arguments followed by a third optional one:

1. `imagePaths`: Paths to our input images.
2. `classNames`: Our `CLASSES` dictionary.
3. `width`: The width that we'll be resizing images to prior to training. Providing a pre-defined width is extremely helpful when working with images with large spatial dimensions. Images that are too large can cause your GPU to run out of memory.

The following function can be used to initiate loading either our training or validation images:

```
83     def load_lesions(self, idxs):
84         # loop over all class names and add each to the 'lesion'
85         # dataset
86         for (classID, label) in self.classNames.items():
87             self.add_class("lesion", classID, label)
88 
89         # loop over the image path indexes
90         for i in idxs:
91             # extract the image filename to serve as the unique
92             # image ID
93             imagePath = self.imagePaths[i]
94             filename = imagePath.split(os.path.sep)[-1]
95 
96             # add the image to the dataset
97             self.add_image("lesion", image_id=filename,
98                 path=imagePath)
```

The `load_lesions` function accepts a single argument, the `idxs` of either the training or validation images (i.e., either `trainIdxs` or `valIdxs`).

Lines 86 and 87 loop over all class unique integer ID and human readable name pairs and adds them to the `lesion` dataset via the `add_class` method.

We then start looping over each of the `idxs` on **Line 90**. For each index `i`, we:

- Grab the corresponding `imagePath`
- Derive the filename from the image path
- Call `add_image`, adding the image to the `lesion` dataset and supplying the unique `image_id` and the path to the image itself.

The Dataset class of Mask R-CNN requires that we define two functions: load_image and load_mask. Both of these functions encapsulate all logic needed to load an input image from disk along with its corresponding mask.

Let's start by examining the load_image function:

```
100     def load_image(self, imageID):
101         # grab the image path, load it, and convert it from BGR to
102         # RGB color channel ordering
103         p = self.image_info[imageID]["path"]
104         image = cv2.imread(p)
105         image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
106 
107         # resize the image, preserving the aspect ratio
108         image = imutils.resize(image, width=self.width)
109 
110         # return the image
111         return image
```

The load_image method accepts a single argument which is our unique imageID. We can lookup all information passed into add_image (back on **Line 97 and 98** in the previous code block) by passing the imageID into the image_info dictionary maintained by the Dataset class.

Line 103 extracts the path to the input image which is then read from disk and converted from BGR to RGB channel ordering on **Lines 104 and 105**.

OpenCV stores images in BGR ordering; however, the Mask R-CNN library assumes RGB ordering, hence the channel reordering. **Line 108** then resizes our image to a fixed width.

Similarly, let's examine our load_mask method:

```
113     def load_mask(self, imageID):
114         # grab the image info and derive the full annotation path
115         # file path
116         info = self.image_info[imageID]
117         filename = info["id"].split(".")[0]
118         annotPath = os.path.sep.join([MASKS_PATH,
119             "{}_segmentation.png".format(filename)])
```

Just like load_image, the load_mask function requires a single argument — the unique image ID.

Our goal in this code block is to (1) take the image filename and (2) derive the corresponding mask file path from it. Since we know all input images have the format isic_<7-digit unique ID>.jpg and all masks have the format isic_<7-digit unique ID>_segmentation.png, we simply need to extract the first portion of the image filename and then append _segmentation.png to it to derive the mask filename (**Lines 116-119**).

We can then create the full annotation path by combining the MASKS_PATH root with the filename.

The next step is to load the actual mask from disk and extract all instances:

```
121         # load the annotation mask and resize it, *making sure* to
122         # use nearest neighbor interpolation
123         annotMask = cv2.imread(annotPath)
124         annotMask = cv2.split(annotMask)[0]
```

```
125             annotMask = imutils.resize(annotMask, width=self.width,
126                 inter=cv2.INTER_NEAREST)
127             annotMask[annotMask > 0] = 1
128
129             # determine the number of unique class labels in the mask
130             classIDs = np.unique(annotMask)
131
132             # the class ID with value '0' is actually the background
133             # which we should ignore and remove from the unique set of
134             # class identifiers
135             classIDs = np.delete(classIDs, [0])
```

Line 123 loads our input image from disk. OpenCV treats each input image it reads as a 3-channel RGB image (but with BGR ordering). Since we know our mask is a grayscale image (implying all R, G, and B channels have the same value), we can split the image into channels and discard all but the first one (**Line 124**).

The annotation mask is then resized to have the same width as our input image (**Lines 125 and 126**), taking care to apply ***nearest neighbor interpolation*** for reasons described in Section 21.1.1 — if you cannot recall why nearest neighbor interpolation is used, stop now and reread Section 21.1.1.

The ISIC 2018 masks have two values: 0 for background and 255 for foreground. However, our CLASSES requires us to have a value of 1 for all lesions, hence we threshold the annotMask, setting all values greater than zero to 1, implying that all pixels with a value of 1 must be a lesion mask.

Line 130 computes the set of unique classIDs in the annotMask which should trivially be 0 (background) and 1 (foreground). Since the background class is unneeded we need to explicitly delete it from the classIDs (**Line 135**).

R If you're attempting to train a Mask R-CNN with more than one class you'll need to modify the above code block, in particular **Line 127** — take care to ensure you can map each and every object in an image to its corresponding class ID.

The final step in the load_mask method is to create a mask for *every object* in the annotation mask:

```
137             # allocate memory for our [height, width, num_instances]
138             # array where each "instance" effectively has its own
139             # "channel" -- since there is only one lesion per image we
140             # know the number of instances is equal to 1
141             masks = np.zeros((annotMask.shape[0], annotMask.shape[1], 1),
142                 dtype="uint8")
```

On **Lines 141 and 142** we allocate memory for a masks array. The masks array should have the same width and height as the annotation mask (the first two dimensions).

The third dimension is the number of objects in the annotMask. For the ISIC 2018 skin lesion boundary detection challenge, the number of objects per image is *always* one. However, for your own dataset, you may have more than one object so you'll want to adjust this value accordingly.

Now that masks array is defined we can loop over all classIDs, which should again trivially be an array with only the value 1 in it:

```
144             # loop over the class IDs
145             for (i, classID) in enumerate(classIDs):
```

```
146                 # construct a mask for *only* the current label
147                 classMask = np.zeros(annotMask.shape, dtype="uint8")
148                 classMask[annotMask == classID] = 1
149 
150                 # store the class mask in the masks array
151                 masks[:, :, i] = classMask
152 
153             # return the mask array and class IDs
154             return (masks.astype("bool"), classIDs.astype("int32"))
```

For each of our unique `classIDs` we allocate memory for a `classMask` — this `mask` will be *binary*, having two values: `0` and `1`.

A value of `0` indicates that a pixel is part of the background. A value of `1` indicates that the pixel is foreground *and* belongs to the current `classID`.

Line 148 finds all (x, y)-coordinates in the `annotMask` with value `classID` and then sets all corresponding (x, y)-coordinates in `classMask` to one. We then store `classMask` in `masks` on **Line 151**.

Our `load_mask` function is *required* to return a 2-tuple on **Line 154**.

The first entry in the tuple is the `masks` array which *must* be returned as type boolean. Keep in mind that our `masks` array is required to have the shape `(width, height, num_instances)`.

The second entry in the tuple *must* have a length of `num_instances` as it maps each entry in `masks` to its corresponding classID. Since the ISIC 2018 dataset (1) consists of just one class label and (2) only one object per image, we can simply return the `classIDs` array.

Before we move on to the rest of the training script (which is *far* easier to comprehend and follow than all previous code blocks), I will say that the `load_mask` function will give you the biggest headaches when attempting to train your own Mask R-CNN.

I'll be providing my suggestions and best practices (including code) when defining both your `load_image` and `load_mask` function later in this chapter, but do keep in mind that training a Mask R-CNN is *less* about the algorithm and *more* about the data preparation.

You'll want to take ***excruciating care*** that `load_image` and your `load_mask` functions are defined correctly, otherwise you'll be left scratching your head looking at confusing error messages or nonsensical training results.

Again, I want to reiterate, these functions are the *lifeblood* of training your Mask R-CNN. *Do not* attempt to train your Mask R-CNN until you are positively certain they are behaving correctly *and* as you intended.

All that said, let's get into the code that can be used to train our Mask R-CNN and make predictions:

```
156 if __name__ == "__main__":
157     # construct the argument parser and parse the arguments
158     ap = argparse.ArgumentParser()
159     ap.add_argument("-m", "--mode", required=True,
160         help="either 'train', 'predict', or 'investigate'")
161     ap.add_argument("-w", "--weights",
162         help="optional path to pretrained weights")
163     ap.add_argument("-i", "--image",
164         help="optional path to input image to segment")
165     args = vars(ap.parse_args())
```

Line 156 checks to see if we are actually executing our script (i.e., typing `python lesions.py ...` in our command line).

Our script requires a single command line argument along with two optional ones:

1. `--mode`: Either `train`, `predict`, or `investigate`. The `train` mode will be responsible for actually training our Mask R-CNN while `predict` mode will be used exclusively for making predictions on input images. We can use the `investigate` mode for debugging and visually validating our dataset prior to training.
2. `--weights`: Path to our trained model weights during `predict` mode.
3. `--image`: Path to our input image during `predict` mode.

Let's go ahead and explore the `train` mode:

```
167     # check to see if we are training the Mask R-CNN
168     if args["mode"] == "train":
169         # load the training dataset
170         trainDataset = LesionBoundaryDataset(IMAGE_PATHS, CLASS_NAMES)
171         trainDataset.load_lesions(trainIdxs)
172         trainDataset.prepare()
173 
174         # load the validation dataset
175         valDataset = LesionBoundaryDataset(IMAGE_PATHS, CLASS_NAMES)
176         valDataset.load_lesions(valIdxs)
177         valDataset.prepare()
178 
179         # initialize the training configuration
180         config = LesionBoundaryConfig()
181         config.display()
```

Lines 170-172 constructs our `trainDataset`. Notice how we instantiate the `LesionBoundaryDataset` by supplying the `IMAGE_PATHS` and `CLASS_NAMES`.

From there, we pass the `trainIdxs` into `load_lesions` to instruct the `LesionBoundaryDataset` to only load the training images. A call to `.prepare` is the final step in preparing the training dataset.

Similarly, we perform the same set of actions on **Lines 175-177**, this time supplying the `valIdxs` to create the `valDataset`.

Line 180 initializes our training configuration. A call to `.display()` prints all configuration options and their corresponding values to our terminal — this information can be very helpful when debugging and tuning hyperparameters.

As mentioned earlier in this section, we will be applying data augmentation when training to both (1) reduce overfitting and (2) increasing the ability of our model to generalize:

```
183         # initialize the image augmentation process
184         aug = iaa.SomeOf((0, 2), [
185             iaa.Fliplr(0.5),
186             iaa.Flipud(0.5),
187             iaa.Affine(rotate=(-10, 10))
188         ])
```

Here we are defining our image augmentation rules. We'll start by picking `SomeOf`, either zero, one or two, of the following operations:

- Flipping horizontally with probability 0.5 (**Line 185**)
- Flipping vertically with probability 0.5 (**Line 186**).
- Randomly rotating between -10 and 10 degrees (**Line 187**).

When training the aug variable will be called and will apply our augmentation rules.
Finally, we are now ready to train our Mask R-CNN:

```
190         # initialize the model and load the COCO weights so we can
191         # perform fine-tuning
192         model = modellib.MaskRCNN(mode="training", config=config,
193             model_dir=LOGS_AND_MODEL_DIR)
194         model.load_weights(COCO_PATH, by_name=True,
195             exclude=["mrcnn_class_logits", "mrcnn_bbox_fc",
196                 "mrcnn_bbox", "mrcnn_mask"])
197 
198         # train *just* the layer heads
199         model.train(trainDataset, valDataset, epochs=20,
200             layers="heads", learning_rate=config.LEARNING_RATE,
201             augmentation=aug)
202 
203         # unfreeze the body of the network and train *all* layers
204         model.train(trainDataset, valDataset, epochs=40,
205             layers="all", learning_rate=config.LEARNING_RATE / 10,
206             augmentation=aug)
```

Lines 192 and 193 instantiates our Mask R-CNN object. Notice how (1) we are explicitly telling the model we are in `training` mode and (2) we pass in our training `config`.

Lines 194-196 load our COCO weights from disk. We'll be fine-tuning this model for instance segmentation on our lesions dataset.

As we learned from Chapter 5 in the *Practitioner Bundle*, we typically fine-tune our newly initialized layer heads *before* training the body of the network.

Lines 199-201 freeze the weights of the body and train *just* the layer heads. Here we supply both our `trainDataset` and `valDataset`, allowing the network to train for 20 epochs. We use the default learning rate of 0.001.

After the 20th epoch we then unfreeze the body and allow the *entire* network to train for an additional 20 epochs (for a total of 40 epochs). The learning rate is lowered by a factor of ten to help reduce overfitting.

Once our model is trained we can move on to the “predict" mode:

```
208     # check to see if we are predicting using a trained Mask R-CNN
209     elif args["mode"] == "predict":
210         # initialize the inference configuration
211         config = LesionBoundaryInferenceConfig()
212 
213         # initialize the Mask R-CNN model for inference
214         model = modellib.MaskRCNN(mode="inference", config=config,
215             model_dir=LOGS_AND_MODEL_DIR)
216 
217         # load our trained Mask R-CNN
218         weights = args["weights"] if args["weights"] \
219             else model.find_last()
220         model.load_weights(weights, by_name=True)
```

Line 211 initializes our *inference* configuration. We then initialize our Mask R-CNN in `inference` mode, supplying the inference configuration as well (**Lines 214 and 215**).

Lines 218-220 load the model weights from disk. We can either *explicitly* supply the path to the model weights via the `--weights` command line argument or we can use the `.find_last()` utility function of the `model` which examines our model snapshot directory and finds the latest model snapshot.

The next step is to prepare our image for prediction:

```
222         # load the input image, convert it from BGR to RGB channel
223         # ordering, and resize the image
224         image = cv2.imread(args["image"])
225         image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
226         image = imutils.resize(image, width=1024)
227 
228         # perform a forward pass of the network to obtain the results
229         r = model.detect([image], verbose=1)[0]
```

We start by loading our input image from disk, and just like our `load_image` function, we reorder the channels from BGR to RGB ordering as well as resize the image to have a fixed with of 1024 pixels.

To obtain the detections and masks from our Mask R-CNN, we simply call the `model.detect` function. The returned value `r` is a dictionary containing four keys: `scores`, `masks`, `class_ids`, and `rois`.

Let's examine how we can use the keys to obtain the masks for each lesion:

```
231         # loop over of the detected object's bounding boxes and
232         # masks, drawing each as we go along
233         for i in range(0, r["rois"].shape[0]):
234             mask = r["masks"][:, :, i]
235             image = visualize.apply_mask(image, mask,
236                 (1.0, 0.0, 0.0), alpha=0.5)
237             image = visualize.draw_box(image, r["rois"][i],
238                 (1.0, 0.0, 0.0))
```

We start by looping over each of the detected bounding boxes (`rois`). **Line 234** extracts the actual `mask` for the current object.

If you examine the shape of the `mask` you will notice it has the same dimensions as the `image`, and furthermore, the mask will have two binary values:

1. `True`: The *foreground* region.
2. `False`: The *background* region.

Lines 235-236 call the `apply_mask` of the `visualize` submodule of Mask R-CNN. Under the hood, `apply_mask` is applying alpha blending to overlay the `mask` on our `image`. **Lines 237 and 238** draw our bounding box on the `image`.

At this point we are going to use OpenCV functions rather than the Mask R-CNN utility functions to draw on our `image`:

```
240         # convert the image back to BGR so we can use OpenCV's
241         # drawing functions
242         image = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)
243 
244         # loop over the predicted scores and class labels
245         for i in range(0, len(r["scores"])):
```

```
246             # extract the bounding box information, class ID, label,
247             # and predicted probability from the results
248             (startY, startX, endY, end) = r["rois"][i]
249             classID = r["class_ids"][i]
250             label = CLASS_NAMES[classID]
251             score = r["scores"][i]
252 
253             # draw the class label and score on the image
254             text = "{}: {:.4f}".format(label, score)
255             y = startY - 10 if startY - 10 > 10 else startY + 10
256             cv2.putText(image, text, (startX, y),
257                 cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 255, 0), 2)
```

Since we'll be using the OpenCV drawing functions, we need to convert our image back to BGR channel ordering (**Line 242**). On **Line 245** we loop over all predicted scores (i.e., probabilities).

It's important to note we don't need to explicitly threshold the scores to filter out weak detections here — that action is already performed for us via the `DETECTION_MIN_CONFIDENCE` value we suppled to the lesion boundary inference configuration.

For each score we extract the bounding box coordinates (**Line 248**) along with the `classID` and `label` (**Lines 249 and 250**). Given our bounding box coordinates, label, and score, we draw them on the `image` (**Lines 254-257**).

At last, we can display our output image to our screen:

```
259         # resize the image so it more easily fits on our screen
260         image = imutils.resize(image, width=512)
261 
262         # show the output image
263         cv2.imshow("Output", image)
264         cv2.waitKey(0)
```

Line 260 resizes our image so that it can easily fit onto our screen while **Lines 263 and 264** actually show the image.

The final mode we are going to look at is used for investigating and debugging your dataset, images, and masks *before* you even start training. I provide more insight into the reasoning behind such a mode existing in Section 21.6 below, so for now, let's look at just the code:

```
273     # check to see if we are investigating our images and masks
274     elif args["mode"] == "investigate":
275         # load the training dataset
276         trainDataset = LesionBoundaryDataset(IMAGE_PATHS, CLASS_NAMES)
277         trainDataset.load_lesions(trainIdxs)
278         trainDataset.prepare()
279 
280         # load the 0-th training image and corresponding masks and
281         # class IDs in the masks
282         image = trainDataset.load_image(0)
283         (masks, classIDs) = trainDataset.load_mask(0)
284 
285         # show the image spatial dimensions which is HxWxC
286         print("[INFO] image shape: {}".format(image.shape))
287 
288         # show the masks shape which should have the same width and
```

```
289             # height of the images but the third dimension should be
290             # equal to the total number of instances in the image itself
291             print("[INFO] masks shape: {}".format(masks.shape))
292 
293             # show the length of the class IDs list along with the values
294             # inside the list -- the length of the list should be equal
295             # to the number of instances dimension in the 'masks' array
296             print("[INFO] class IDs length: {}".format(len(classIDs)))
297             print("[INFO] class IDs: {}".format(classIDs))
```

Lines 269-271 load our training dataset, just as if we were going to train our network. However, instead of training, we load the 0-th image and corresponding mask from the dataset.

Line 279 displays the spatial dimensions of the `image`. The actual spatial dimensions are *height* $\times$ *width* $\times$ *channels*.

Make sure these values are what you expect. For example, if you are resizing your images to have a fixed width or height, make sure that particular dimension has been properly resized. Similarly, if you're working with RGB images, make sure the number of channels is three.

Line 284 displays the dimensions of the `mask` volume which will have the shape *height* $\times$ *width* $\times$ *num_instances*. The `mask` volume should have the *same* width and height as the `image` volume; however, the third dimension should be equal to the total number of objects/instances in the image itself.

We then investigate the `classIDs` array on **Lines 289 and 290**. The length of the `classIDs` array should be equal to the total number of instances in the `masks` array (i.e., the third dimension).

You should also verify that the `classIDs` array contains only the unique label IDs for the objects in `masks` — the array should *not* contain a value of `0` as zero is reserved for the background.

The final step I recommend when investigating your dataset is to visually validate that the Mask R-CNN "thinks" your dataset is constructed properly.

To accomplish such as task we are going to loop over a randomly sampled set of indexes, loading the image and corresponding mask, and then use the `display_top_masks` helper utility to actually visualize the objects on our screen:

```
292             # determine a sample of training image indexes and loop over
293             # them
294             for i in np.random.choice(trainDataset.image_ids, 3):
295                 # load the image and masks for the sampled image
296                 print("[INFO] investigating image index: {}".format(i))
297                 image = trainDataset.load_image(i)
298                 (masks, classIDs) = trainDataset.load_mask(i)
299 
300                 # visualize the masks for the current image
301                 visualize.display_top_masks(image, masks, classIDs,
302                     trainDataset.class_names)
```

The output of running the above code block can be seen in Figure 21.6. Here you can see that the Mask R-CNN framework and dataset utilities have been able to correctly read our training data — our input image is displayed to the screen and the "lesion" region is correctly masked.

I recommend you *always* perform such debugging *prior* to training your Mask R-CNN as it will save you significantly headaches down the line. I provide more tips and suggestions to successfully train Mask R-CNNs on your own dataset in Section 21.6, but for now, let's move on to training the actual network.

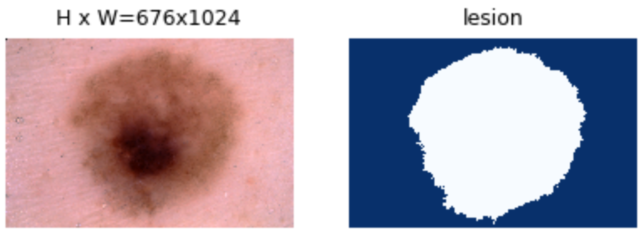


Figure 21.6: An example output of running `visualize.display_top_masks`. On the *left* is our input image. On the *right* is our corresponding mask. You should use the `investigate` mode to visually validate that the Keras + Mask R-CNN framework is properly parsing your dataset.

21.4.3 Training Your Mask R-CNN

That was quite a lot of code to review! Let's see how we can actually train our Mask R-CNN on the ISIC 2018 dataset. Open up a terminal and execute the following command:

```
$ python lesions.py --mode train
...
Starting at epoch 0. LR=0.001
Epoch 1/20
2075/2075 - 1234s 595ms/step - loss: 1.1190 - val_loss: 0.8835
Epoch 2/20
2075/2075 - 1192s 575ms/step - loss: 0.8643 - val_loss: 0.8789
Epoch 3/20
2075/2075 - 1176s 567ms/step - loss: 0.8274 - val_loss: 0.7446
...
```

Here you can see that our Mask R-CNN is starting to train. Each epoch is taking approximately 19 minutes on my Titan X GPU. Loss steadily decreases throughout training the layer heads.

Then, at the end of epoch 20, we unfreeze all layers and allow the *entire* network to be trained:

```
Starting at epoch 20. LR=0.0001
Epoch 21/40
2075/2075 - 1757s 847ms/step - loss: 0.4888 - val_loss: 0.5687
Epoch 22/40
2075/2075 - 1757s 847ms/step - loss: 0.4409 - val_loss: 0.6123
...
Epoch 39/40
2075/2075 - 1770s 853ms/step - loss: 0.2854 - val_loss: 0.5199
Epoch 40/40
2075/2075 - 1867s 900ms/step - loss: 0.2739 - val_loss: 0.5220
```

Notice that now we are training the entire network that epochs are now taking approximately 30 minutes on my Titan X. Again, both training and validation loss continue to drop but at the end of the training process we see loss is starting to saturate at 0.5, indicating that we likely cannot reduce loss further without additional hyperparameter adjustment.

21.5 Mask Predictions with Your Mask R-CNN

Now that our Mask R-CNN is trained, let's see how we can make predictions with it. Open up your terminal and execute the following command:

```
$ python lesions.py --mode predict \
    --image isic2018/ISIC2018_Task1-2_Training_Input/ISIC_0000000.jpg
```

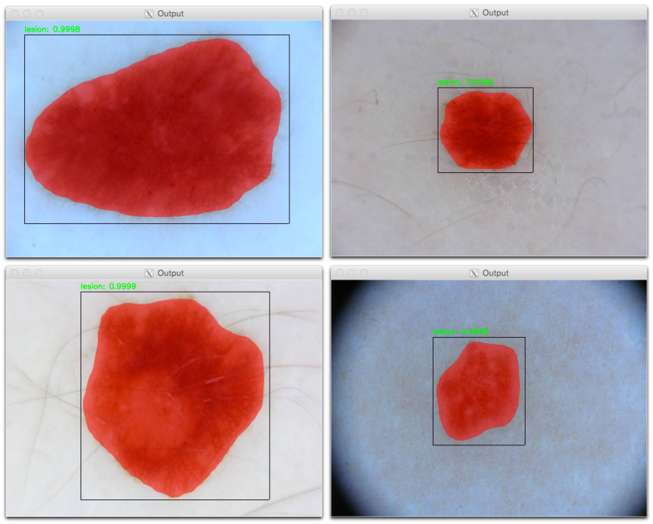


Figure 21.7: Sample predictions from our Mask R-CNN applied to skin lesion boundary detection. Notice how accurate our pixel-wise masks are.

I have included a montage of example mask predictions in Figure 21.7. Notice how our Mask R-CNN is doing an *extremely* good job at predicting the pixel-wise mask of the lesion.

Such a model could easily be used in a skin cancer prediction pipeline where the first step is detecting and extracting the region ROI. Further downstream steps of the pipeline could include classifying and identifying the skin lesion type (i.e., Task #3 of the ISIC 2018 challenge).

21.6 Tips When Training Your Own Mask R-CNN

Inevitably, you *will* run into problems when training your own Mask R-CNN, similar to how you likely ran into issues when training your first Faster R-CNN or SSD in Chapter 15 and 17 [REF] of the *ImageNet Bundle*.

Keep in mind that running into problems is the *norm* when working with state-of-the-art computer vision and deep learning libraries — if such computer vision solutions existed off the shelf it would be highly unlikely you would be reading this book in the first place.

In this section I'll detail my suggestions when training your own Mask R-CNN and how to overcome errors I personally encountered (and you likely will as well).

21.6.1 Training Never Starts

When I first started working with the `mrcnn` package I noticed a strange, intermittent behavior where my training script would get "stuck" at the start of Epoch #1 but never actually *start* training:

```
$ python lesions.py --mode train
...
Starting at epoch 0. LR=0.001
Epoch 1/20
...training never starts...
```

In fact, this behavior *never* occurred when working with the pills dataset in the case study in the following chapter, but *did* happen intermittently with the skin lesions dataset.

Instead of throwing my hands in the air and giving up, I first sought out the Matterport Mask R-CNN GitHub Issues page (http://pyimg.co/dr978). I then used the "search" feature to look for keywords related to "training hanging" or "training not starting". Through my short 2-3 minute research I came across a handful of separate threads that detailed the issue.

While the actual cause of the problem was never properly determined, the issue could be resolved by opening up the Mask_RCNN/mrcnn/model.py file and finding the self.keras_model.fit_generator call:

```
2365        self.keras_model.fit_generator(
2366            train_generator,
2367            initial_epoch=self.epoch,
2368            epochs=epochs,
2369            steps_per_epoch=self.config.STEPS_PER_EPOCH,
2370            callbacks=callbacks,
2371            validation_data=val_generator,
2372            validation_steps=self.config.VALIDATION_STEPS,
2373            max_queue_size=100,
2374            workers=workers,
2375            use_multiprocessing=True,
```

If you are on the same mrcnn branch as me, this code block should start around **Line 2364**. You'll want to then set use_multiprocessing=False — this fix allowed my Mask R-CNN to start training. The GitHub Issues threads I referenced above also said that setting workers=0 was also necessary but I did not need to update the workers value.

The important takeaway here is that *you need to do your research* when you encounter an error.

There will ***absolutely*** be errors you encounter — that's part of the challenge in doing state-of-the-art computer vision and deep learning work. It is *your job* to do the necessary investigative work in trying to troubleshoot and resolve the problem. It may seem like a pain but this type of research and investigative work is a *necessary* and *critical* skill for you to master.

Keep at it, don't get discouraged, and do the work — it will all work out in the end.

21.6.2 Focus On Your Image and Mask Loading Functions

I cannot stress enough that you need to debug, re-debug, and re-re-debug your load_image and load_mask functions. The load_mask function in particular will be the one that gives you the most headaches.

Training a Mask R-CNN is *less* about the actual mrcnn library and *more* about the image and mask loading utilities. You need to double, triple, and quadruple check these functions *before* you even start training. **Don't rush into training, doing so will only cause you problems later!**

When implementing your own load_image and load_mask functions, make sure you refer to my investigate mode in the lesions.py file. Make sure you're calling both load_image and load_mask for a given set of images.

From there, you'll want to examine the shape and values of the returned NumPy arrays. Ask yourself:

1. Are my input image and mask the same spatial dimensions?
2. Have I remembered to convert from RGB to BGR channel ordering?
3. Are there multiple objects in each image and mask?
4. Can the multiple objects be *different* classes?
5. Have I defined my `masks` array with the proper number of instance as the final dimension?
6. How am I "finding" each individual object in the mask? Array indexing? A special lookup key that is provided with the dataset?
7. Have I ensured that the final dimensions of the `masks` array is the same length as the class identifiers I am returning from `load_mask`?

Use this set of questions and checklist as a starting point and then *add your own questions and items to the list* as you get more experienced with training Mask R-CNNs.

Once you've verified your `load_image` and `load_masks` functions are behaving as you expect, you'll want to visually validate your assumptions using the `display_top_masks` function (detailed in Section 21.4.2).

Visually validating that `mrcnn` can *correctly* parse out each object and corresponding mask should be an *absolute requirement* before trying to train your Mask R-CNN.

21.6.3 Refer to the Mask R-CNN Samples

Matterport, the maintainer of the Keras + Mask R-CNN implementation, has a number of super helpful samples of (1) loading your own custom data and (2) training a Mask R-CNN in their `samples` directory of the repository: http://pyimg.co/isgoe.

Inside the samples you'll find Jupyter Notebooks to guide you through various training examples. I personally found these examples *extremely* helpful and informative, *especially* when working with undocumented features.

I would *highly encourage* you to work through these examples as the educational experience is *well worth* the investment of your time.

21.7 Summary

In this chapter we discussed the differences between object detection, instance segmentation, and semantic segmentation. When performing object detection, our goal is to find all objects in an input image, label the objects, and provide the bounding box (x,y)-coordinates of each object.

Instance segmentation takes object detection a step further, providing not only the bounding box coordinates, *but also a pixel-wise mask of the object*, allowing us to segment the object from the image.

Semantic segmentation builds on instance segmentation but requires us to label *all pixels in an image*, thereby assigning a label to each pixel. Semantic segmentation is most commonly applied in "scene understanding" applications, such as self-driving cars.

Semantic segmentation algorithms are challenging to say the least, and furthermore, there most applications require only instance segmentation — we chose to study instance segmentation in this chapter.

To facilitate our understanding of instance segmentation, we reviewed the Mask R-CNN algorithm [71] which builds on the Faster R-CNN work by Girshick et al. [42] We then used the Keras Mask R-CNN implementation [72] to train our own custom instance segmentation model on the ISIC 2018 dataset [82, 83] — this model is capable of localizing skin lesion regions for cancer detection with high accuracy.

The ISIC 2018 dataset was pre-compiled for us though. How might we go about *labeling and annotating our own custom dataset*, followed by training a Mask R-CNN on our custom dataset? To answer that question, you’ll need to refer to the next chapter.