11. Convolutional Neural Networks

Our entire review of machine learning and neural networks thus far have been leading up to this point: ***understanding Convolutional Neural Networks (CNNs)*** and the role they play in deep learning.

In traditional feedforward neural networks (like the ones we studied in Chapter 10), each neuron in the input layer is connected to every output neuron in the next layer – we call this a *fully-connected* (FC) layer. However, in CNNs, we don't use FC layers until the *very last layer(s)* in the network. We can thus define a CNN as a neural network that swaps in a specialized "convolutional" layer in place of "fully-connected" layer for at least *one* of the layers in the network [10].

A nonlinear activation function, such as ReLU, is then applied to the output of these convolutions and the process of convolution => activation continues (along with a mixture of other layer types to help reduce the width and height of the input volume and help reduce overfitting) until we finally reach the end of the network and apply one or two FC layers where we can obtain our final output classifications.

Each layer in a CNN applies a different set of filters, typically hundreds or thousands of them, and combines the results, feeding the output into the next layer in the network. During training, **a CNN automatically learns the values for these filters**.

In the context of image classification, our CNN may learn to:

- Detect edges from raw pixel data in the first layer.
- Use these edges to detect shapes (i.e., "blobs") in the second layer.
- Use these shapes to detect higher-level features such as facial structures, parts of a car, etc. in the highest layers of the network.

The last layer in a CNN uses these higher-level features to make predictions regarding the contents of the image. In practice, CNNs give us two key benefits: *local invariance* and *compositionality*. The concept of *local invariance* allows us to classify an image as containing a particular object *regardless* of where in the image the object appears. We obtain this local invariance through the usage of "pooling layers" (discussed later in this chapter) which identifies regions of our input volume with a high response to a particular filter.

The second benefit is compositionality. Each filter composes a local patch of lower-level features

into a higher-level representation, similar to how we can compose a set of mathematical functions that build on the output of previous functions: $f(g(x(h(x))))$ – this composition allows our network to learn more rich features deeper in the network. For example, our network may build edges from pixels, shapes from edges, and then complex objects from shapes – all in an automated fashion that happens *naturally* during the training process. The concept of building higher-level features from lower-level ones is exactly why CNNs are so powerful in computer vision.

In the rest of this chapter, we'll discuss exactly *what* convolutions are and the role they play in deep learning. We'll then move on to the building blocks of CNNs: layers, and the various types of layers you'll use to build your own CNNs. We'll wrap up this chapter by looking at common patterns that are used to stack these building blocks to create CNN architectures that perform well on a diverse set of image classification tasks.

After reviewing this chapter, we'll have (1) a strong understanding of Convolutional Neural Networks and the thought process that goes into building one and (2) a number of CNN "recipes" we can use to construct our own network architectures. In our next chapter, we'll use these fundamentals and recipes to train CNNs of our own.

11.1 Understanding Convolutions

In this section, we'll address a number of questions, including:

- *What* are image convolutions?
- *What* do they do?
- *Why* do we use them?
- *How* do we apply them to images?
- **And what role do convolutions play in deep learning?**

The word "convolution" sounds like a fancy, complicated term – but it's really not. If you have any prior experience with computer vision, image processing, or OpenCV before, you've *already* applied convolutions, ***whether you realize it or not!***

Ever apply *blurring* or *smoothing* to an image? Yep, that's a convolution. What about *edge detection*? Yup, convolution. Have you opened Photoshop or GIMP to *sharpen* an image? You guessed it – convolution. Convolutions are one of the most *critical, fundamental building-blocks* in computer vision and image processing.

But the term itself tends to scare people off – in fact, on the surface, the word even appears to have a negative connotation (why would anyone want to "convolute" something?) Trust me, convolutions are anything but scary. They're actually quite easy to understand.

In terms of deep learning, an (image) ***convolution is an element-wise multiplication of two matrices followed by a sum***.

Seriously. That's it. *You just learned what a convolution is:*

1. Take two matrices (which both have the same dimensions).
2. Multiply them, element-by-element (i.e., *not* the dot product, just a simple multiplication).
3. Sum the elements together.

We'll learn more about convolutions, kernels, and how they are used inside CNNs in the remainder of this section.

11.1.1 Convolutions versus Cross-correlation

A reader with prior background in computer vision and image processing may have identified my description of a *convolution* above as a *cross-correlation* operation instead. Using cross-correlation instead of convolution is actually by design. Convolution (denoted by the $\star$ operator) over a

two-dimensional input image I and two-dimensional kernel K is defined as:

$$S(i,j) = (I \star K)(i,j) = \sum_m \sum_n K(i-m, j-n) I(m,n) \tag{11.1}$$

However, nearly all machine learning and deep learning libraries use the simplified *cross-correlation* function

$$S(i,j) = (I \star K)(i,j) = \sum_m \sum_n K(i+m, j+n) I(m,n) \tag{11.2}$$

All this math amounts to is a sign change in how we access the coordinates of the image I (i.e., we don't have to "flip" the kernel relative to the input when applying cross-correlation).

Again, many deep learning libraries use the simplified cross-correlation operation and call it convolution – **we will use the same terminology here**. For readers interested in learning more about the mathematics behind convolution vs. cross-correlation, please refer to Chapter 3 of *Computer Vision: Algorithms and Applications* by Szelski [119].

11.1.2 The "Big Matrix" and "Tiny Matrix" Analogy

An image is a *multidimension matrix*. Our image has a width (# of columns) and height (# of rows), just like a matrix. But unlike traditional matrices you have worked with back in grade school, images also have a *depth* to them – the number of *channels* in the image.

For a standard RGB image, we have a depth of 3 – one channel for *each* of the Red, Green, and Blue channels, respectively. Given this knowledge, we can think of an image as *big matrix* and a *kernel* or *convolutional matrix* as a *tiny matrix* that is used for blurring, sharpening, edge detection, and other processing functions. Essentially, this *tiny* kernel sits on top of the *big* image and slides from left-to-right and top-to-bottom, applying a mathematical operation (i.e., a *convolution*) at each (x,y)-coordinate of the original image.

It's normal to hand-define kernels to obtain various image processing functions. In fact, you might already be familiar with blurring (average smoothing, Gaussian smoothing, median smoothing, etc.), edge detection (Laplacian, Sobel, Scharr, Prewitt, etc.), and sharpening – *all* of these operations are forms of hand-defined kernels that are *specifically designed* to perform a particular function.

So that raises the question: *is there a way to* ***automatically*** *learn these types of filters?* And even use these filters for *image classification* and *object detection?* **You bet there is.** But before we get there, we need to understand kernels and convolutions a bit more.

11.1.3 Kernels

Again, let's think of an image as a *big matrix* and a kernel as a *tiny matrix* (at least in respect to the original "big matrix" image), depicted in Figure 11.1. As the figure demonstrates, we are sliding the kernel (red region) from left-to-right and top-to-bottom along the original image. At each (x,y)-coordinate of the original image, we stop and examine the neighborhood of pixels located at the **center** of the image kernel. We then take this neighborhood of pixels, *convolve* them with the kernel, and obtain a single output value. The output value is stored in the output image at the same (x,y)-coordinates as the center of the kernel.

If this sounds confusing, no worries, we'll be reviewing an example in the next section. But before we dive into an example, let's take a look at what a kernel looks like (Figure 11.3):

$$K = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{11.3}$$

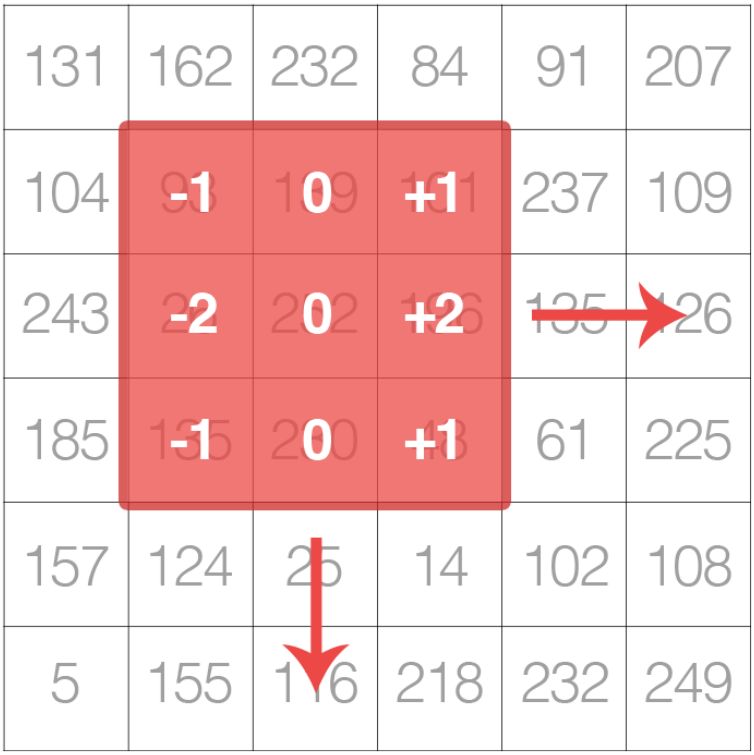

Figure 11.1: A kernel can be visualized as a small matrix that slides across, from left-to-right and top-to-bottom, of a larger image. At each pixel in the input image, the neighborhood of the image is convolved with the kernel and the output stored.

Above we have defined a square 3×3 kernel (any guesses on what this kernel is used for?). Kernels can be of arbitrary rectangular size MxN, provided that **both** M and N are ***odd integers***.

R Most kernels applied to deep learning and CNNs are $N \times N$ *square* matrices, allowing us to take advantage of optimized linear algebra libraries that operate most efficiently on square matrices.

We use an *odd* kernel size to ensure there is a valid integer (x, y)-coordinate at the center of the image (Figure 11.2). On the *left*, we have a 3×3 matrix. The center of the matrix is located at $x = 1, y = 1$ where the top-left corner of the matrix is used as the origin and our coordinates are zero-indexed. But on the *right,* we have a 2×2 matrix. The center of this matrix would be located at $x = 0.5, y = 0.5$.

But as we know, without applying interpolation, there is no such thing as pixel location $(0.5, 0.5)$ – our pixel coordinates must be integers! This reasoning is exactly why we use *odd* kernel sizes: to always ensure there is a valid (x, y)-coordinate at the center of the kernel.

11.1.4 A Hand Computation Example of Convolution

Now that we have discussed the basics of kernels, let's discuss the actual convolution operation and see an example of it actually being applied to help us solidify our knowledge. In image processing, a convolution requires three components:

1. An input image.
2. A kernel matrix that we are going to apply to the input image.

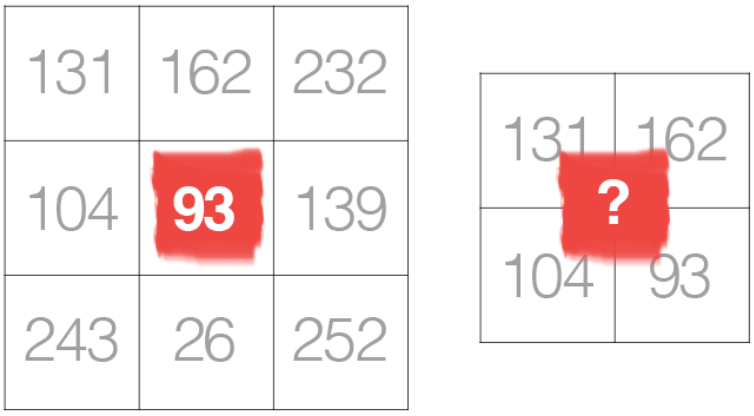

Figure 11.2: **Left:** The center pixel of a 3×3 kernel is located at coordinate $(1,1)$ (highlighted in red). **Right:** What is the center coordinate of a kernel of size 2×2?

3. An output image to store the output of the image convolved with the kernel.

Convolution (i.e., cross-correlation) is actually very easy. All we need to do is:

1. Select an (x,y)-coordinate from the original image.
2. Place the **center** of the kernel at this (x,y)-coordinate.
3. Take the element-wise multiplication of the input image region and the kernel, then sum up the values of these multiplication operations into a single value. The sum of these multiplications is called the **kernel output**.
4. Use the same (x,y)-coordinates from **Step #1**, but this time, store the kernel output at the same (x,y)-location as the output image.

Below you can find an example of convolving (denoted mathematically as the $\star$ operator) a 3×3 region of an image with a 3×3 kernel used for blurring:

$$O_{i,j} = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \star \begin{bmatrix} 93 & 139 & 101 \\ 26 & 252 & 196 \\ 135 & 230 & 18 \end{bmatrix} = \begin{bmatrix} 1/9 \times 93 & 1/9 \times 139 & 1/9 \times 101 \\ 1/9 \times 26 & 1/9 \times 252 & 1/9 \times 196 \\ 1/9 \times 135 & 1/9 \times 230 & 1/9 \times 18 \end{bmatrix} \tag{11.4}$$

Therefore,

$$O_{i,j} = \sum \begin{bmatrix} 10.3 & 15.4 & 11.2 \\ 2.8 & 28.0 & 21.7 \\ 15.0 & 25.5 & 2.0 \end{bmatrix} \approx 132. \tag{11.5}$$

After applying this convolution, we would set the pixel located at the coordinate (i,j) of the output image O to $O_{i,j} = 132$.

That's all there is to it! Convolution is simply the sum of element-wise matrix multiplication between the kernel and neighborhood that the kernel covers of the input image.

11.1.5 Implementing Convolutions with Python

To help us further understand the concept of convolutions, let's look at some actual code that will reveal how kernels and convolutions are implemented. This source code will not only help you understand *how* to apply convolutions to images, but also enable you to understand *what's going on under the hood* when training CNNs.

Open up a new file, name it `convolutions.py`, and let's get to work:

```
1  # import the necessary packages
2  from skimage.exposure import rescale_intensity
3  import numpy as np
4  import argparse
5  import cv2
```

We start on **Lines 2-5** by importing our required Python packages. We'll be using NumPy and OpenCV for our standard numerical array processing and computer vision functions, along with the scikit-image library to help us implement our own custom convolution function.

Next, we can start defining this `convolve` method:

```
7  def convolve(image, K):
8      # grab the spatial dimensions of the image and kernel
9      (iH, iW) = image.shape[:2]
10     (kH, kW) = K.shape[:2]
11
12     # allocate memory for the output image, taking care to "pad"
13     # the borders of the input image so the spatial size (i.e.,
14     # width and height) are not reduced
15     pad = (kW - 1) // 2
16     image = cv2.copyMakeBorder(image, pad, pad, pad, pad,
17         cv2.BORDER_REPLICATE)
18     output = np.zeros((iH, iW), dtype="float")
```

The `convolve` function requires two parameters: the (grayscale) `image` that we want to convolve with `kernel`. Given both our `image` and `kernel` (which we presume to be NumPy arrays), we then determine the spatial dimensions (i.e., width and height) of each (**Lines 10 and 11**).

Before we continue, it's important to understand the process of "sliding" a convolutional matrix across an image, applying the convolution, and then storing the output, which will actually *decrease* the spatial dimensions of our input image. Why is this?

Recall that we "center" our computation around the center (x, y)-coordinate of the input image that the kernel is currently positioned over. *This positioning implies there is no such thing as "center" pixels for pixels that fall along the border of the image* (as the corners of the kernel would be "hanging off" the image where the values are undefined), depicted by Figure 11.3.

The decrease in spatial dimension is simply a side effect of applying convolutions to images. Sometimes this effect is desirable, and other times it is not, it simply depends on your application.

However, in most cases, we want our *output image* to have the *same dimensions as our input image*. To ensure the dimensions are the same, we apply ***padding*** (**Lines 15-18**). Here we are simply replicating the pixels along the border of the image, such that the output image will match the dimensions of the input image.

Other padding methods exist, including *zero padding* (filling the borders with zeros – very common when building Convolutional Neural Networks) and *wrap around* (where the border pixels are determined by examining the opposite side of the image). In most cases, you will see either replicate or zero padding. Replicate padding is more commonly used when aesthetics are concerned while zero padding is best for efficiency.

We are now ready to apply the actual convolution to our image:

```
20     # loop over the input image, "sliding" the kernel across
21     # each (x, y)-coordinate from left-to-right and top-to-bottom
```

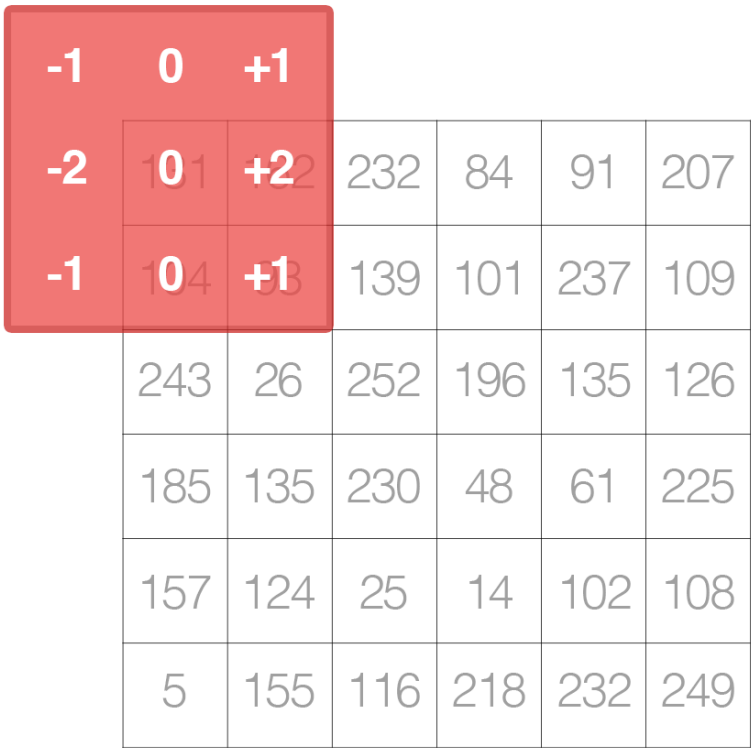

Figure 11.3: If we attempted to apply convolution at the pixel located at $(0,0)$, then our 3×3 kernel would "hang off" off the edge of the image. Notice how there are no input image pixel values for the first row and first column of the kernel. Because of this, we always either (1) start convolution at the first valid position or (2) apply zero padding (covered later in this chapter).

```
22      for y in np.arange(pad, iH + pad):
23          for x in np.arange(pad, iW + pad):
24              # extract the ROI of the image by extracting the
25              # *center* region of the current (x, y)-coordinates
26              # dimensions
27              roi = image[y - pad:y + pad + 1, x - pad:x + pad + 1]
28
29              # perform the actual convolution by taking the
30              # element-wise multiplication between the ROI and
31              # the kernel, then summing the matrix
32              k = (roi * K).sum()
33
34              # store the convolved value in the output (x, y)-
35              # coordinate of the output image
36              output[y - pad, x - pad] = k
```

Lines 22 and 23 loop over our `image`, "sliding" the kernel from left-to-right and top-to-bottom, one pixel at a time. **Line 27** extracts the Region of Interest (ROI) from the `image` using NumPy array slicing. The `roi` will be centered around the current (x,y)-coordinates of the `image`. The `roi` will also have the same size as our `kernel`, which is critical for the next step.

Convolution is performed on **Line 32** by taking the element-wise multiplication between the `roi` and `kernel`, followed by summing the entries in the matrix. The output value `k` is then stored

in the `output` array at the same (x,y)-coordinates (relative to the input image).

We can now finish up our `convolve` method:

```
38          # rescale the output image to be in the range [0, 255]
39          output = rescale_intensity(output, in_range=(0, 255))
40          output = (output * 255).astype("uint8")
41
42          # return the output image
43          return output
```

When working with images, we typically deal with pixel values falling in the range $[0,255]$. However, when applying convolutions, we can easily obtain values that fall *outside* this range. In order to bring our `output` image back into the range $[0,255]$, we apply the `rescale_intensity` function of scikit-image (**Line 39**).

We also convert our image back to an unsigned 8-bit integer data type on **Line 40** (previously, the `output` image was a floating point type in order to handle pixel values outside the range $[0,255]$). Finally, the `output` image is returned to the calling function on **Line 43**.

Now that we've defined our `convolve` function, let's move on to the driver portion of the script. This section of our program will handle parsing command line arguments, defining a series of kernels we are going to apply to our image, and then displaying the output results:

```
45  # construct the argument parse and parse the arguments
46  ap = argparse.ArgumentParser()
47  ap.add_argument("-i", "--image", required=True,
48      help="path to the input image")
49  args = vars(ap.parse_args())
```

Our script requires only a single command line argument, `--image`, which is the path to our input image. We can then define two kernels used for blurring and smoothing an image:

```
51  # construct average blurring kernels used to smooth an image
52  smallBlur = np.ones((7, 7), dtype="float") * (1.0 / (7 * 7))
53  largeBlur = np.ones((21, 21), dtype="float") * (1.0 / (21 * 21))
```

To convince yourself that this kernel is performing blurring, notice how each entry in the kernel is an *average* of $1/S$ where S is the total number of entries in the matrix. Thus, this kernel will multiply each input pixel by a small fraction and take the sum – this is exactly the definition of the average.

We then have a kernel responsible for sharpening an image:

```
55  # construct a sharpening filter
56  sharpen = np.array((
57      [0, -1, 0],
58      [-1, 5, -1],
59      [0, -1, 0]), dtype="int")
```

Then the Laplacian kernel used to detect edge-like regions:

```
# construct the Laplacian kernel used to detect edge-like
# regions of an image
laplacian = np.array((
	[0, 1, 0],
	[1, -4, 1],
	[0, 1, 0]), dtype="int")
```

The Sobel kernels can be used to detect edge-like regions along both the x and y axis, respectively:

```
# construct the Sobel x-axis kernel
sobelX = np.array((
	[-1, 0, 1],
	[-2, 0, 2],
	[-1, 0, 1]), dtype="int")

# construct the Sobel y-axis kernel
sobelY = np.array((
	[-1, -2, -1],
	[0, 0, 0],
	[1, 2, 1]), dtype="int")
```

And finally, we define the emboss kernel:

```
# construct an emboss kernel
emboss = np.array((
	[-2, -1, 0],
	[-1, 1, 1],
	[0, 1, 2]), dtype="int")
```

Explaining how each of these kernels were formulated is outside the scope of this book, so for the time being simply understand that these are kernels that were *manually built* to perform a given operation.

For a thorough treatment of how kernels are mathematically constructed and proven to perform a given image processing operation, please refer to Szeliksi (Chapter 3) [119]. I also recommend using this excellent kernel visualization tool from Setosa.io [120].

Given all these kernels, we can lump them together into a set of tuples called a "kernel bank":

```
# construct the kernel bank, a list of kernels we're going to apply
# using both our custom `convole` function and OpenCV's `filter2D`
# function
kernelBank = (
	("small_blur", smallBlur),
	("large_blur", largeBlur),
	("sharpen", sharpen),
	("laplacian", laplacian),
	("sobel_x", sobelX),
	("sobel_y", sobelY),
	("emboss", emboss))
```

Constructing this list of kernels enables use to loop over them and visualize their output in an efficient manner, as the code block below demonstrates:

```
98  # load the input image and convert it to grayscale
99  image = cv2.imread(args["image"])
100 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
101
102 # loop over the kernels
103 for (kernelName, K) in kernelBank:
104     # apply the kernel to the grayscale image using both our custom
105     # 'convolve' function and OpenCV's 'filter2D' function
106     print("[INFO] applying {} kernel".format(kernelName))
107     convolveOutput = convolve(gray, K)
108     opencvOutput = cv2.filter2D(gray, -1, K)
109
110     # show the output images
111     cv2.imshow("Original", gray)
112     cv2.imshow("{} - convole".format(kernelName), convolveOutput)
113     cv2.imshow("{} - opencv".format(kernelName), opencvOutput)
114     cv2.waitKey(0)
115     cv2.destroyAllWindows()
```

Lines 99 and 100 load our image from disk and convert it to grayscale. Convolution operators can and are applied to RGB or other multi-channel volumes, but for the sake of simplicity, we'll only apply our filters to grayscale images.

We start looping over our set of kernels in the `kernelBank` on **Line 103** and then apply the current `kernel` to the `gray` image on **Line 104** by calling our function `convolve` method, defined earlier in the script.

As a sanity check, we also call `cv2.filter2D` which also applies our `kernel` to the `gray` image. The `cv2.filter2D` function is OpenCV's much more optimized version of our `convolve` function. The main reason I am including both here is for us to sanity check our custom implementation.

Finally, **Lines 111-115** display the output images to our screen for each kernel type.

Convolution Results

To run our script (and visualize the output of various convolution operations), just issue the following command:

```
$ python convolutions.py --image jemma.png
```

You'll then see the results of applying the `smallBlur` kernel to the input image in Figure 11.4. On the *left*, we have our original image. Then, in the *center*, we have the results from the `convolve` function. And on the *right*, the results from `cv2.filter2D`. A quick visual inspection will reveal that our output matches `cv2.filter2D`, indicating that our `convolve` function is working properly. Furthermore, our image now appears "blurred" and "smoothed", thanks to the smoothing kernel.

Let's apply a larger blur, results of which can be seen in Figure 11.5 (*top-left*). This time I am omitting the `cv2.filter2D` results to save space. Comparing the results from Figure 11.5 to Figure 11.4, notice how as the size of the averaging kernel *increases*, the amount of blurring in the output image *increases* as well.

We can also sharpen our image (Figure 11.5, *top-mid*) and detect edge-like regions via the Laplacian operator (*top-right*).

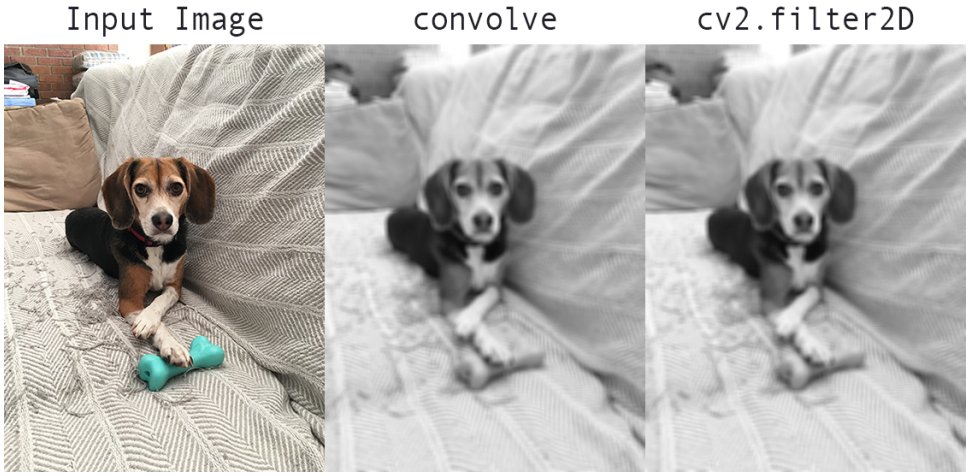

Figure 11.4: **Left:** Our original input image. **Center:** Applying a 7×7 average blur using our custom `convolve` function. **Right:** Applying the same 7×7 blur using OpenCV's `cv2.filter2D` – notice how the output of the two functions is identical, implying that our `convolve` method is implemented correctly.

The `sobelX` kernel is used to find vertical edges in the image (Figure 11.5, *bottom-left*), while the `sobelY` kernel reveals horizontal edges (*bottom-mid*). Finally, we can see the result of the emboss kernel in the *bottom-left*.

11.1.6 The Role of Convolutions in Deep Learning

As you've gathered from this section, we must *manually hand-define* each of our kernels for each of our various image processing operations, such as smoothing, sharpening, and edge detection. That's all fine and good, **but what if there was a way to *learn* these filters instead?**

Is it possible to define a machine learning algorithm that can look at our input images and eventually *learn* these types of operators? In fact, there is – these types of algorithms are the primary focus of this book: **Convolutional Neural Networks (CNNs)**.

By applying convolutions filters, nonlinear activation functions, pooling, and backpropagation, CNNs are able to learn filters that can detect edges and blob-like structures in lower-level layers of the network – and then use the edges and structures as "building blocks", eventually detecting high-level objects (e.x., faces, cats, dogs, cups, etc.) in the deeper layers of the network.

This process of using the lower-level layers to learn high-level features is exactly the *compositionality* of CNNs that we were referring to earlier. But exactly *how* do CNNs do this? The answer is by stacking a specific set of layers in a purposeful manner. In our next section, we'll discuss these types of layers, followed by examining common layer stacking patterns that are widely used among many image classification tasks.

11.2 CNN Building Blocks

As we learned from Chapter 10, neural networks accept an input image/feature vector (one input node for each entry) and transform it through a series of hidden layers, commonly using nonlinear activation functions. Each hidden layer is also made up of a set of neurons, where each neuron is *fully-connected* to all neurons in the previous layer. The last layer of a neural network (i.e., the "output layer") is also fully-connected and represents the final output classifications of the network.

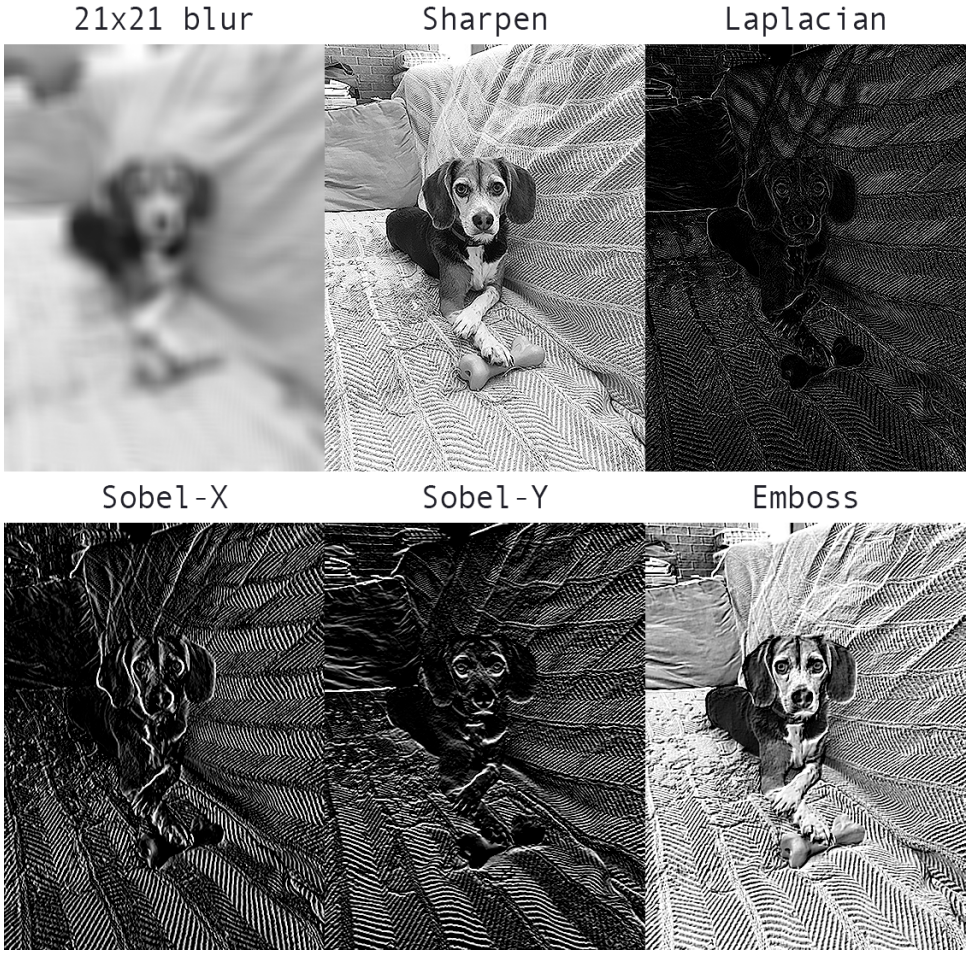

Figure 11.5: **Top-left:** Applying a 21×21 average blur. Notice how this image is more blurred than in Figure 11.4. **Top-mid:** Using a sharpening kernel to enhance details. **Top-right:** Detecting edge-like operations via the Laplacian operator. **Bottom-left:** Computing vertical edges using the Sobel-X kernel. **Bottom-mid:** Finding horizontal edges using the Sobel-Y kernel. **Bottom-right:** Applying an emboss kernel.

However, as the results of Section 10.1.4 demonstrate, neural networks operating directly on raw pixel intensities:

1. Do not scale well as the image size increases.
2. Leaves much accuracy to be desired (i.e., a standard feedforward enroll network on CIFAR-10 obtained only 15% accuracy).

To demonstrate how standard neural networks do not scale well as image size increases, let's again consider the CIFAR-10 dataset. Each image in CIFAR-10 is 32×32 with a Red, Green, and Blue channel, yielding a total of $32 \times 32 \times 3 = 3,072$ total inputs to our network.

A total of $3,072$ inputs does not seem to amount to much, but consider if we were using 250×250 pixel images – the total number of inputs and weights would jump to $250 \times 250 \times 3 = 187,500$ – and this number is only for the input layer alone! Surely, we would want to add multiple hidden layers with varying number of nodes per lay – these parameters can quickly add up, and given the poor performance of standard neural networks on raw pixel intensities, this bloat is hardly worth it.

Instead, we can use *Convolutional Neural Networks (CNNs)* that take advantage of the input

image structure and define a network architecture in a more sensible way. Unlike a standard neural network, layers of a CNN are arranged in a *3D volume* in three dimensions: **width**, **height**, and **depth** (where *depth* refers to the third dimension of the volume, such as the number of channels in an image or the number of filters in a layer).

To make this example more concrete, again consider the CIFAR-10 dataset: the input volume will have dimensions $32 \times 32 \times 3$ (width, height, and depth, respectively). Neurons in subsequent layers will only be connected to a *small region* of the layer before it (rather than the fully-connected structure of a standard neural network) – we call this ***local connectivity*** which enables us to save a *huge* amount of parameters in our network. Finally, the output layer will be a $1 \times 1 \times N$ volume which represents the image distilled into a single vector of class scores. In the case of CIFAR-10, given ten classes, $N = 10$, yielding a $1 \times 1 \times 10$ volume.

11.2.1 Layer Types

There are many types of layers used to build Convolutional Neural Networks, but the ones you are most likely to encounter include:

- Convolutional (`CONV`)
- Activation (`ACT` or `RELU`, where we use the same of the actual activation function)
- Pooling (`POOL`)
- Fully-connected (`FC`)
- Batch normalization (`BN`)
- Dropout (`DO`)

Stacking a series of these layers in a specific manner yields a CNN. We often use simple text diagrams to describe a CNN: `INPUT => CONV => RELU => FC => SOFTMAX`

Here we define a simple CNN that accepts an input, applies a convolution layer, then an activation layer, then a fully-connected layer, and, finally, a softmax classifier to obtain the output classification probabilities. The `SOFTMAX` activation layer is often omitted from the network diagram as it is assumed it directly follows the final `FC`.

Of these layer types, `CONV` and `FC`, (and to a lesser extent, `BN`) are the only layers that contain parameters that are *learned* during the training process. Activation and dropout layers are not considered true "layers" themselves, but are often included in network diagrams to make the architecture *explicitly* clear. Pooling layers (`POOL`), of equal importance as `CONV` and `FC`, are also included in network diagrams as they have a *substantial impact* on the spatial dimensions of an image as it moves through a CNN.

`CONV`, `POOL`, `RELU`, and `FC` are the most important when defining your actual network architecture. That's not to say that the other layers are not critical, but take a backseat to this critical set of four as they define the *actual architecture itself*.

Activation functions themselves are practically **assumed** to be part of the architecture, When defining CNN architectures we often `omit` the activation layers from a table/diagram to save space; however, the activation layers are *implicitly* assumed to be part of the architecture.

In the remainder of this section, we'll review each of these layer types in detail and discuss the parameters associated with each layer (and how to set them). Later in this chapter I'll discuss in more detail how to stack these layers properly to build your own CNN architectures.

11.2.2 Convolutional Layers

The `CONV` layer is the core building block of a Convolutional Neural Network. The `CONV` layer parameters consist of a set of K learnable filters (i.e., "kernels"), where each filter has a width and a height, and are nearly always square. These filters are small (in terms of their spatial dimensions) but extend throughout the full depth of the volume.

For inputs to the CNN, the depth is the number of channels in the image (i.e., a depth of three when working with RGB images, one for each channel). For volumes deeper in the network, the depth will be the number of filters applied in the *previous* layer.

To make this concept more clear, let's consider the forward-pass of a CNN, where we convolve each of the K filters across the width and height of the input volume, just like we did in Section 11.1.5 above. More simply, we can think of each of our K kernels sliding across the input region, computing an element-wise multiplication, summing, and then storing the output value in a 2-dimensional *activation map*, such as in Figure 11.6.

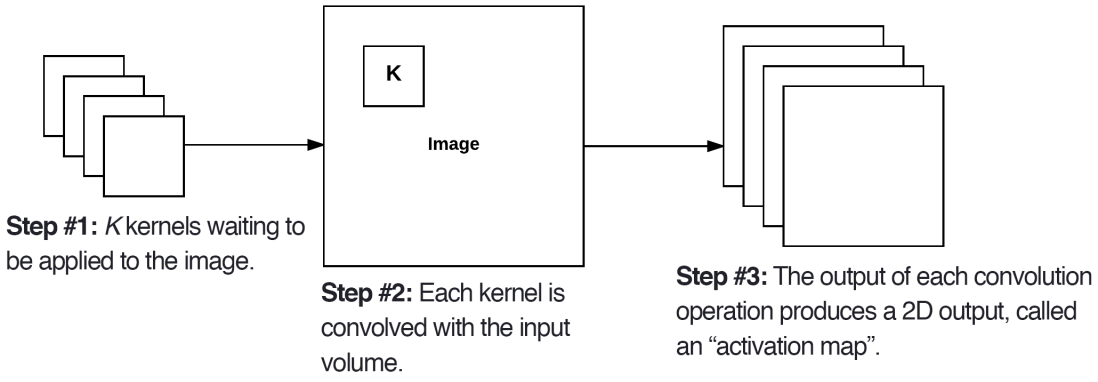

Figure 11.6: **Left:** At each convolutional layer in a CNN, there are K kernels applied to the input volume. **Middle:** Each of the K kernels is convolved with the input volume. **Right:** Each kernel produces an 2D output, called an *activation map*.

After applying all K filters to the input volume, we now have K, 2-dimensional activation maps. We then stack our K activation maps along the depth dimension of our array to form the final output volume (Figure 11.7).

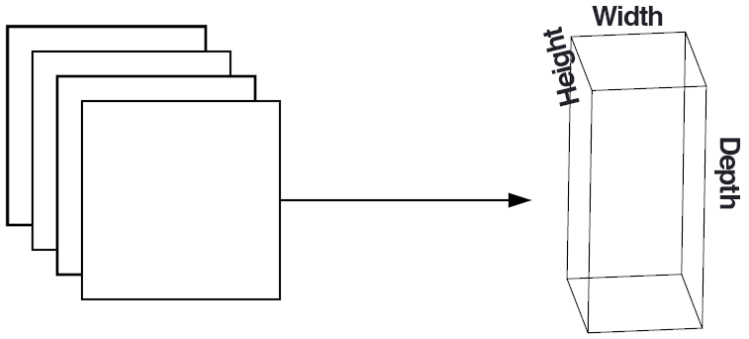

Figure 11.7: After obtaining the K activation maps, they are stacked together to form the input volume to the next layer in the network.

Every entry in the output volume is thus an output of a neuron that "looks" at only a small region of the input. In this manner, the network "learns" filters that activate when they see a specific type of feature at a *given spatial location* in the input volume. In lower layers of the network, filters may activate when they see edge-like or corner-like regions.

Then, in the deeper layers of the network, filters may activate in the presence of high-level features, such as parts of the face, the paw of a dog, the hood of a car, etc. This activation concept

goes back to our neural network analogy in Chapter 10 – these neurons are becoming "excited" and "activating" when they see a particular pattern in an input image.

The concept of convolving a small filter with a large(r) input volume has special meaning in Convolutional Neural Networks – specifically, the **local connectivity** and the **receptive field** of a neuron. When working with images, it's often impractical to connect neurons in the current volume to *all* neurons in the previous volume – there are simply too many connections and too many weights, making it impossible to train deep networks on images with large spatial dimensions. Instead, when utilizing CNNs, we choose to connect each neuron to only a *local region* of the input volume – we call the size of this local region the **receptive field** (or simply, the variable F) of the neuron.

To make this point clear, let's return to our CIFAR-10 dataset where the input volume as an input size of $32 \times 32 \times 3$. Each image thus has a width of 32 pixels, a height of 32 pixels, and a depth of 3 (one for each RGB channel). If our receptive field is of size 3×3, then each neuron in the `CONV` layer will connect to a 3×3 local region of the image for a total of $3 \times 3 \times 3 = 27$ weights (remember, the depth of the filters is three because they extend through the full depth of the input image, in this case, three channels).

Now, let's assume that the spatial dimensions of our input volume have been reduced to a smaller size, but our depth is now larger, due to utilizing more filters deeper in the network, such that the volume size is now $16 \times 16 \times 94$. Again, if we assume a receptive field of size 3×3, then every neuron in the `CONV` layer will have a total of $3 \times 3 \times 94 = 846$ connections to the input volume. Simply put, the receptive field F is the ***size*** of the filter, yielding an $F \times F$ kernel that is convolved with the input volume.

At this point we have explained the connectivity of neurons in the input volume, but not the arrangement or size of the output volume. There are three parameters that control the size of an output volume: the **depth**, **stride**, and **zero-padding** size, each of which we'll review below.

Depth

The *depth* of an output volume controls the number of neurons (i.e., filters) in the `CONV` layer that connect to a local region of the input volume. Each filter produces an activation map that "activate" in the presence of oriented edges or blobs or color.

For a given `CONV` layer, the depth of the activation map will be K, or simply the number of filters we are learning in the current layer. The set of filters that are "looking at" the same (x, y) location of the input is called the **depth column**.

Stride

Consider Figure 11.1 earlier in this chapter where we described a convolution operation as "sliding" a small matrix across a large matrix, stopping at each coordinate, computing an element-wise multiplication and sum, then storing the output. This description is similar to a *sliding window* (http://pyimg.co/0yizo) that slides from left-to-right and top-to-bottom across an image.

In the context of Section 11.1.5 on convolution above, we only took a step of one pixel each top. In the context of CNNs, the same principle can be applied – for each step, we create a new depth column around the local region of the image where we convolve each of the K filters with the region and store the output in a 3D volume. When creating our `CONV` layers we normally use a stride step size S of either $S = 1$ or $S = 2$.

Smaller strides will lead to overlapping receptive fields and larger output volumes. Conversely, larger strides will result in less overlapping receptive fields and smaller output volumes. To make the concept of convolutional stride more concrete, consider the Table 11.1 where we have a 5×5 input image (*left*) along with a 3×3 Laplacian kernel (right).

Using $S = 1$, our kernel slides from left-to-right and top-to-bottom, one pixel at a time, producing the following output (Figure 11.2, *left*). However, if we were to apply the same operation, only

95	242	186	152	39
39	14	220	153	180
5	247	212	54	46
46	77	133	110	74
156	35	74	93	116

0	1	0
1	-4	1
0	1	0

Table 11.1: Our input 5×5 image *(left)* that we are going to convolve with a Laplacian kernel *(right)*.

692	-315	-6
-680	-194	305
153	-59	-86

692	-6
153	-86

Table 11.2: **Left:** Output of convolution with 1×1 stride. **Right:** Output of convolution with 2×2 stride. Notice how a larger stride can reduce the spatial dimensions of the input.

this time with a stride of $S = 2$, we skip *two pixels at a time* (two pixels along the x-axis and two pixels along the y-axis), producing a smaller output volume (*right*).

Thus, we can see how convolution layers can be used to reduce the spatial dimensions of the input volumes simply by changing the stride of the kernel. As we'll see later in this section, convolutional layers and pooling layers are the primary methods to reduce spatial input size. The pooling layers section will also provide a more visual example of how vary stride sizes will affect output size.

Zero-padding

As we know from Section 11.1.5, we need to "pad" the borders of an image to retain the *original image size* when applying a convolution – the same is true for filters inside of a CNN. Using zero-padding, we can "pad" our input along the borders such that our output volume size matches our input volume size. The amount of padding we apply is controlled by the parameter P.

This technique is *especially critical* when we start looking at deep CNN architectures that apply multiple `CONV` filters on top of each other. To visualize zero-padding, again refer to Table 11.1 where we applied a 3×3 Laplacian kernel to a 5×5 input image with a stride of $S = 1$.

We can see in Table 11.3 (*left*) how the output volume is *smaller* (3×3) than the input volume (5×5) due to the nature of the convolution operation. If we instead set $P = 1$, we can pad our input volume with zeros (*middle*) to create a 7×7 volume and then apply the convolution operation, leading to an output volume size that matches the original input volume size of 5×5 (*right*).

Without zero padding, the spatial dimensions of the input volume would decrease too quickly, and we wouldn't be able to train deep networks (as the input volumes would be too tiny to learn any useful patterns from).

Putting all these parameters together, we can compute the size of an output volume as a function of the input volume size (W, assuming the input images are square, which they nearly always are), the receptive field size F, the stride S, and the amount of zero-padding P. To construct a valid `CONV` layer, we need to ensure the following equation is an integer:

$$((W - F + 2P)/S) + 1 \tag{11.6}$$

If it is *not* an integer, then the strides are set incorrectly, and the neurons cannot be tiled such that they fit across the input volume in a symmetric way.

692	-315	-6
-680	-194	305
153	-59	-86

0	0	0	0	0	0	0
0	95	242	186	152	39	0
0	39	14	220	153	180	0
0	5	247	212	54	46	0
0	46	77	133	110	74	0
0	156	35	74	93	116	0
0	0	0	0	0	0	0

-99	-673	-130	-230	176
-42	692	-315	-6	-482
312	-680	-194	305	124
54	153	-59	-86	-24
-543	167	-35	-72	-297

Table 11.3: **Left:** The output of applying a 3×3 convolution to a 5×5 output (i.e., the spatial dimensions decrease). **Right:** Applying zero-padding to the original input with $P = 1$ increases the spatial dimensions to 7×7. **Bottom:** After applying the 3×3 convolution to the padded input, our output volume times matches the *original* input volume size of 5×5, thus zero-padding helps us preserve spatial dimensions.

As an example, consider the first layer of the AlexNet architecture which won the 2012 ImageNet classification challenge and is *hugely* responsible for the current boom of deep learning applied to image classification. Inside their paper, Krizhevsky et al. [94] documented their CNN architecture according to Figure 11.8.

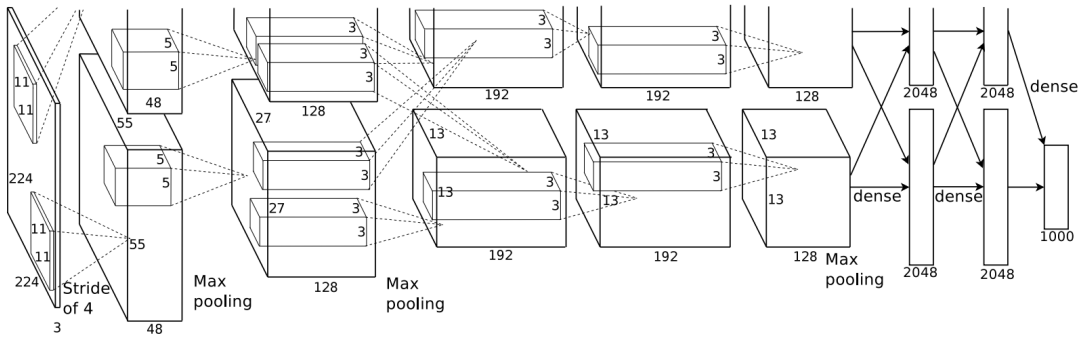

Figure 11.8: The original AlexNet architecture diagram provided by Krizhevsky et al. [94] Notice how the input image is documented to be $224 \times 224 \times 3$, although this cannot be possible due to Equation 11.6. It's also worth noting that we are unsure why the top-half of this figure is cutoff from the original publication.

Notice how the first layer claims that the input image size is 224×224 pixels. However, this can't possibly be correct if we apply our equation above using 11×11 filters, a stride of four, and no padding:

$$((224 - 11 + 2(0))/4) + 1 = 54.25 \tag{11.7}$$

Which is certainly not an integer.

For novice readers just getting started in deep learning and CNNs, this small error in such a seminal paper has caused countless errors of confusion and frustration. It's unknown why this typo occurred, but it's likely that Krizhevsky et al. used 227×227 input images, since:

$$((227 - 11 + 2(0))/4) + 1 = 55 \tag{11.8}$$

Errors like these are more common than you might think, so when implementing CNNs from publications, be sure to *check the parameters yourself* rather than simply assuming the parameters listed are correct. Due to the vast number of parameters in a CNN, it's quite easy to make a typographical mistake when documenting an architecture (I've done it myself many times).

To summarize, the `CONV` layer in the same, elegant manner as Karpathy [121]:

- Accepts an input volume of size $W_{input} \times H_{input} \times D_{input}$ (the input sizes are normally square, so it's common to see $W_{input} = H_{input}$).
- Requires four parameters:
  1. The number of filters K (which controls the *depth* of the output volume).
  2. The receptive field size F (the size of the K kernels used for convolution and is nearly always *square*, yielding an $F \times F$ kernel).
  3. The stride S.
  4. The amount of zero-padding P.
- The output of the `CONV` layer is then $W_{output} \times H_{output} \times D_{output}$, where:
  - $W_{output} = ((W_{input} - F + 2P)/S) + 1$
  - $H_{output} = ((H_{input} - F + 2P)/S) + 1$
  - $D_{output} = K$

We'll review common settings for these parameters in Section 11.3.1 below.

11.2.3 Activation Layers

After each `CONV` layer in a CNN, we apply a nonlinear activation function, such as ReLU, ELU, or any of the other Leaky ReLU variants mentioned in Chapter 10. We typically denote activation layers as `RELU` in network diagrams as since ReLU activations are most commonly used, we may also simply state `ACT` – in either case, we are making it clear that an activation function is being applied inside the network architecture.

Activation layers are not technically "layers" (due to the fact that no parameters/weights are learned inside an activation layer) and are sometimes omitted from network architecture diagrams as it's *assumed* that an activation *immediately follows* a convolution.

In this case, authors of publications will mention which activation function they are using after each `CONV` layer somewhere in their paper. As an example, consider the following network architecture: `INPUT => CONV => RELU => FC`.

To make this diagram more concise, we could simply remove the `RELU` component since it's assumed that an activation always follows a convolution: `INPUT => CONV => FC`. I personally do not like this and choose to *explicitly* include the activation layer in a network diagram to make it clear *when* and *what* activation function I am applying in the network.

An activation layer accepts an input volume of size $W_{input} \times H_{input} \times D_{input}$ and then applies the given activation function (Figure 11.9). Since the activation function is applied in an element-wise manner, the output of an activation layer is always the same as the input dimension, $W_{input} = W_{output}$, $H_{input} = H_{output}$, $D_{input} = D_{output}$.

11.2.4 Pooling Layers

There are two methods to reduce the size of an input volume – `CONV` layers with a stride > 1 (which we've already seen) and `POOL` layers. It is common to insert `POOL` layers in-between consecutive

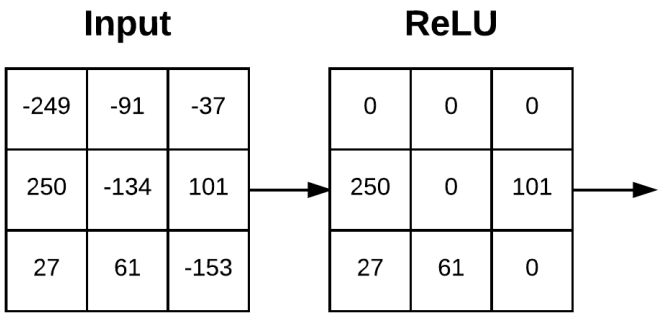

Figure 11.9: An example of an input volume going through a ReLU activation, $max(0,x)$. Activations are done *in-place* so there is no need to create a separate output volume although it is easy to visualize the flow of the network in this manner.

`CONV` layers in a CNN architectures:

`INPUT => CONV => RELU => POOL => CONV => RELU => POOL => FC`

The primary function of the `POOL` layer is to progressively reduce the spatial size (i.e., width and height) of the input volume. Doing this allows us to reduce the amount of parameters and computation in the network – pooling also helps us control overfitting.

`POOL` layers operate on each of the depth slices of an input *independently* using either the *max* or *average* function. Max pooling is typically done in the middle of the CNN architecture to reduce spatial size, whereas average pooling is normally used as the final layer of the network (e.x., GoogLeNet, SqueezeNet, ResNet) where we wish to avoid using `FC` layers entirely. The most common type of `POOL` layer is max pooling, although this trend is changing with the introduction of more exotic micro-architectures.

Typically we'll use a pool size of 2×2, although deeper CNNs that use larger input images (> 200 pixels) may use a 3×3 pool size early in the network architecture. We also commonly set the stride to either $S = 1$ or $S = 2$. Figure 11.10 (heavily inspired by Karpathy et al. [121]) follows an example of applying max pooling with 2×2 pool size and a stride of $S = 1$. Notice for every 2×2 block, we keep only the largest value, take a single step (like a sliding window), and apply the operation again – thus producing an output volume size of 3×3.

We can further decrease the size of our output volume by increasing the stride – here we apply $S = 2$ to the same input (Figure 11.10, *bottom*). For every 2×2 block in the input, we keep only the largest value, then take a step of *two pixels*, and apply the operation again. This pooling allows us to reduce the width and height by a factor of two, effectively discarding 75% of activations from the previous layer.

In summary, `POOL` layers Accept an input volume of size $W_{input} \times H_{input} \times D_{input}$. They then require two parameters:

- The receptive field size F (also called the "pool size").
- The stride S.

Applying the `POOL` operation yields an output volume of size $W_{output} \times H_{output} \times D_{output}$, where:

- $W_{output} = ((W_{input} - F)/S) + 1$
- $H_{output} = ((H_{input} - F)/S) + 1$
- $D_{output} = D_{input}$

In practice, we tend to see two types of max pooling variations:

- **Type #1:** $F = 3, S = 2$ which is called *overlapping pooling* and normally applied to images/input volumes with large spatial dimensions.

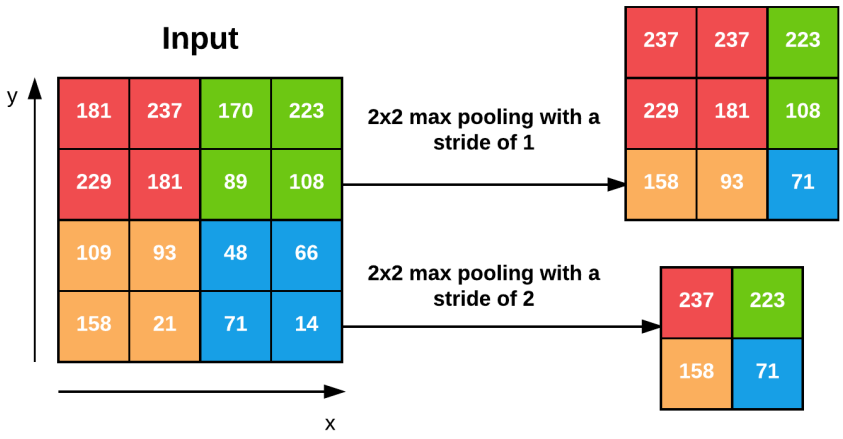

Figure 11.10: **Left:** Our input 4×4 volume. **Right:** Applying 2×2 max pooling with a stride of $S = 1$. **Bottom:** Applying 2×2 max pooling with $S = 2$ – this dramatically reduces the spatial dimensions of our input.

- **Type #2:** $F = 2, S = 2$ which is called *non-overlapping pooling*. This is the most common type of pooling and is applied to images with smaller spatial dimensions.

For network architectures that accept smaller input images (in the range of $32 - 64$ pixels) you may also see $F = 2, S = 1$ as well.

To POOL or CONV?

In their 2014 paper, *Striving for Simplicity: The All Convolutional Net*, Springenberg et al. [122] recommend discarding the `POOL` layer *entirely* and simply relying on `CONV` layers with a larger stride to handle downsampling the spatial dimensions of the volume. Their work demonstrated this approach works very well on a variety of datasets, including CIFAR-10 (small images, low number of classes) and ImageNet (large input images, 1,000 classes). This trend continues with the ResNet architecture [96] which uses `CONV` layers for downsampling as well.

It's becoming increasingly more common to *not* use `POOL` layers in the middle of the network architecture and *only* use average pooling at the end of the network if `FC` layers are to be avoided. Perhaps in the future there won't be pooling layers in Convolutional Neural Networks – but in the meantime, it's important that we study them, learn how they work, and apply them to our own architectures.

11.2.5 Fully-connected Layers

Neurons in `FC` layers are fully-connected to all activations in the previous layer, as is the standard for feedforward neural networks that we've been discussing in Chapter 10. `FC` layers are *always* placed at the end of the network (i.e., we don't apply a `CONV` layer, then an `FC` layer, followed by another `CONV`) layer.

It's common to use one or two `FC` layers prior to applying the softmax classifier, as the following (simplified) architecture demonstrates:

```
INPUT => CONV => RELU => POOL => CONV => RELU => POOL => FC => FC
```

Here we apply two fully-connected layers before our (implied) softmax classifier which will compute our final output probabilities for each class.

11.2.6 Batch Normalization

First introduced by Ioffe and Szgedy in their 2015 paper, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift* [123], batch normalization layers (or `BN` for short), as the name suggests, are used to normalize the activations of a given input volume before passing it into the next layer in the network.

If we consider x to be our mini-batch of activations, then we can compute the normalized $\hat{x}$ via the following equation:

$$\hat{x_i} = \frac{x_i - \mu_\beta}{\sqrt{\sigma_\beta^2 + \varepsilon}} \tag{11.9}$$

During *training*, we compute the μ_β and σ_β over each mini-batch β, where:

$$\mu_\beta = \frac{1}{M}\sum_{i=1}^{m} x_i \qquad\qquad \sigma_\beta^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\beta)^2 \tag{11.10}$$

We set ε equal to a small positive value such as 1e-7 to avoid taking the square root of zero. Applying this equation implies that the activations leaving a batch normalization layer will have approximately zero mean and unit variance (i.e., zero-centered).

At *testing* time, we replace the mini-batch μ_β and σ_β with *running averages* of μ_β and σ_β computed during the training process. This ensures that we can pass images through our network and still obtain accurate predictions without being biased by the μ_β and σ_β from the final mini-batch passed through the network at training time.

Batch normalization has been shown to be *extremely effective* at reducing the number of epochs it takes to train a neural network. Batch normalization also has the added benefit of helping "stabilize" training, allowing for a larger variety of learning rates and regularization strengths. Using batch normalization doesn't alleviate the need to tune these parameters of course, but it *will* make your life easier by making learning rate and regularization less volatile and more straightforward to tune. You'll also tend to notice *lower final loss* and a *more stable loss curve* when using batch normalization in your networks.

The biggest drawback of batch normalization is that it can actually slow down the wall time it takes to train your network (even though you'll need fewer epochs to obtain reasonable accuracy) by 2-3x due to the computation of per-batch statistics and normalization.

That said, I recommend using batch normalization in *nearly every situation* as it does make a significant difference. As we'll see later in this book, applying batch normalization to our network architectures can help us prevent overfitting and allows us to obtain significantly higher classification accuracy in fewer epochs compared to the same network architecture *without* batch normalization.

So, Where Do the Batch Normalization Layers Go?

You've probably noticed in my discussion of batch normalization I've left out exactly *where* in the network architecture we place the batch normalization layer. According to the original paper by Ioffe and Szegedy [123], they placed their batch normalization (`BN`) *before* the activation:

> *"We add the BN transform immediately before the nonlinearity, by normalizing* $x = Wu + b$*."*

Using this scheme, a network architecture utilizing batch normalization would look like this:

```
INPUT => CONV => BN => RELU ...
```

However, this view of batch normalization doesn't make sense from a statistical point of view. In this context, a BN layer is normalizing the distribution of features coming out of a CONV layer. Some of these features may be negative, in which they will be clamped (i.e., set to zero) by a nonlinear activation function such as ReLU.

If we normalize *before* activation, we are essentially including the negative values inside the normalization. Our zero-centered features are then passed through the ReLU where we kill of any activations less than zero (which include features which may have not been negative *before* the normalization) – this layer ordering entirely defeats the purpose of applying batch normalization in the first place.

Instead, if we place the batch normalization *after* the ReLU we will normalize the positive valued features without statistically biasing them with features that would have otherwise not made it to the next CONV layer. In fact, François Chollet, the creator and maintainer of Keras confirms this point stating that the BN should come after the activation:

> *"I can guarantee that recent code written by Christian [Szegedy, from the BN paper] applies relu before BN. It is still occasionally a topic of debate, though."* [124]

It is unclear why Ioffe and Szegedy suggested placing the BN layer before the activation in their paper, but further experiments [125] as well as anecdotal evidence from other deep learning researchers [126] confirm that placing the batch normalization layer *after* the nonlinear activation yields higher accuracy and lower loss in nearly all situations.

Placing the BN after the activation in a network architecture would look like this:

```
INPUT => CONV => RELU => BN ...
```

I can confirm that in nearly all experiments I've performed with CNNs, placing the BN after the RELU yields slightly higher accuracy and lower loss. That said, take note of the word *"nearly"* – there have been a *very small* number of situations where placing the BN before the activation worked better, which implies that you should default to placing the BN after the activation, but may want to dedicate (at most) one experiment to placing the BN before the activation and noting the results.

After running a few of these experiments, you'll quickly realize that BN after the activation performs better and there are more important parameters to your network to tune to obtain higher classification accuracy. I discuss this in more detail in Section 11.3.2 later in this chapter.

11.2.7 Dropout

The last layer type we are going to discuss is dropout. Dropout is actually a form of *regularization* that aims to help prevent overfitting by increasing testing accuracy, perhaps at the expense of training accuracy. For each mini-batch in our training set, dropout layers, with probability p, randomly disconnect inputs from the preceding layer to the next layer in the network architecture.

Figure 11.11 visualizes this concept where we randomly disconnect with probability $p = 0.5$ the connections between two FC layers for a given mini-batch. Again, notice how half of the connections are severed for this mini-batch. After the forward and backward pass are computed for the mini-batch, we re-connect the dropped connections, and then sample another set of connections to drop.

The reason we apply dropout is to reduce overfitting by *explicitly* altering the network architecture at training time. Randomly dropping connections ensures that no single node in the network is

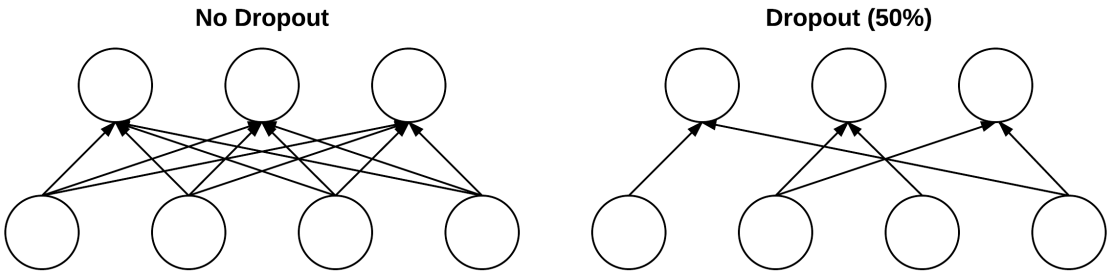

Figure 11.11: **Left:** Two layers of a neural network that are fully-connected with no dropout. **Right:** The same two layers after dropping 50% of the connections.

responsible for "activating" when presented with a given pattern. Instead, dropout ensures there are *multiple, redundant nodes* that will activate when presented with similar inputs – this in turn helps our model to *generalize*.

It is most common to place dropout layers with $p = 0.5$ ***in-between*** `FC` layers of an architecture where the final `FC` layer is assumed to be our softmax classifier:

```
... CONV => RELU => POOL => FC => DO => FC => DO => FC
```

However, as I discuss in Section 11.3.2, we may also apply dropout with smaller probabilities (i.e., $p = 0.10 - 0.25$) in earlier layers of the network as well (normally following a downsampling operation, either via max pooling or convolution).

11.3 Common Architectures and Training Patterns

As we have seen throughout this chapter, Convolutional Neural Networks are made up of four primary layers: `CONV`, `POOL`, `RELU`, and `FC`. Taking these layers and stacking them together in a particular pattern yields a *CNN architecture*.

The `CONV` and `FC` layers (and `BN`) are the only layers of the network that actually learn parameters – the other layers are simply responsible for performing a given operation. Activation layers, (`ACT`) such as `RELU` and dropout aren't technically layers, but are often included in the CNN architecture diagrams to make the operation order *explicitly clear* – we'll adopt the same convention in this section as well.

11.3.1 Layer Patterns

By far, the most common form of CNN architecture is to stack a few `CONV` and `RELU` layers, following them with a `POOL` operation. We repeat this sequence until the volume width and height is small, at which point we apply one or more `FC` layers. Therefore, we can derive the most common CNN architecture using the following pattern [121]:

```
INPUT => [[CONV => RELU]*N => POOL?]*M => [FC => RELU]*K => FC
```

Here the $*$ operator implies one or more and the `?` indicates an optional operation.
Common choices for each reputation include:

- $0 <= N <= 3$
- $M >= 0$
- $0 <= K <= 2$

Below we can see some examples of CNN architectures that follow this pattern:

- `INPUT => FC`
- `INPUT => [CONV => RELU => POOL] * 2 => FC => RELU => FC`
- `INPUT => [CONV => RELU => CONV => RELU => POOL] * 3 => [FC => RELU] * 2 => FC`

Here is an example of a very shallow CNN with only one `CONV` layer ($N = M = K = 0$) which we will review in Chapter 12:

```
INPUT => CONV => RELU => FC
```

Below is an example of an AlexNet-like [94] CNN architecture which has multiple `CONV => RELU => POOL` layer sets, followed by `FC` layers:

```
INPUT => [CONV => RELU => POOL] * 2 => [CONV => RELU] * 3 => POOL =>
     [FC => RELU => DO] * 2 => SOFTMAX
```

For deeper network architectures, such as VGGNet [95], we'll stack two (or more) layers before every `POOL` layer:

```
INPUT => [CONV => RELU] * 2 => POOL => [CONV => RELU] * 2 => POOL =>
     [CONV => RELU] * 3 => POOL => [CONV => RELU] * 3 => POOL =>
     [FC => RELU => DO] * 2 => SOFTMAX
```

Generally, we apply deeper network architectures when we (1) have lots of labeled training data and (2) the classification problem is sufficiently challenging. Stacking multiple `CONV` layers before applying a `POOL` layer allows the `CONV` layers to develop more complex features before the destructive pooling operation is performed.

As we'll discover in the *ImageNet Bundle* of this book, there are more "exotic" network architectures that deviate from these patterns and, in turn, have created patterns of their own. Some architectures remove the `POOL` operation entirely, relying on `CONV` layers to downsample the volume – then, at the end of the network, average pooling is applied rather than `FC` layers to obtain the input to the softmax classifiers.

Network architectures such as GoogLeNet, ResNet, and SqueezeNet [96, 97, 127] are great examples of this pattern and demonstrate how removing `FC` layers leads to less parameters and faster training time.

These types of network architectures also "stack" and concatenate filters across the channel dimension: GoogLeNet applies 1×1, 3×3, and 5×5 filters and then concatenates them together across the channel dimension to learn multi-level features. Again, these architectures are considered more "exotic" and considered advanced techniques.

If you're interested in these more advanced CNN architectures, please refer to the *ImageNet Bundle*; otherwise, you'll want to stick with the basic layer stacking patterns until you learn the fundamentals of deep learning.

11.3.2 Rules of Thumb

In this section, I'll review common rules of thumb when constructing your own CNNs. To start, the images presented to the **input layer** should be *square*. Using square inputs allows us to take advantage of linear algebra optimization libraries. Common input layer sizes include 32×32,

64×64, 96×96, 224×224, 227×227 and 229×229 (leaving out the number of channels for notational convenience).

Secondly, the input layer should also be *divisible by two multiple times* after the first CONV operation is applied. You can do this by tweaking your filter size and stride. The "divisible by two rule" enables the spatial inputs in our network to be conveniently down sampled via POOL operation in an efficient manner.

In general, your CONV layers should use smaller filter sizes such as 3×3 and 5×5. Tiny 1×1 filters are used to learn local features, but only in your more advanced network architectures. Larger filter sizes such as 7×7 and 11×11 may be used as the *first* CONV layer in the network (to reduce spatial input size, provided your images are sufficiently larger than $> 200 \times 200$ pixels); however, after this initial CONV layer the filter size should drop dramatically, otherwise you will reduce the spatial dimensions of your volume too quickly.

You'll also commonly use a stride of $S = 1$ for CONV layers, at least for smaller spatial input volumes (networks that accept larger input volumes that use a stride $S >= 2$ in the first CONV layer). Using a stride of $S = 1$ enables our CONV layers to learn filters while the POOL layer is responsible for downsampling. However, keep in mind that not *all* network architectures follow this pattern – some architectures skip max pooling altogether and rely on the CONV stride to reduce volume size.

My personal preference is to apply **zero-padding** to my CONV layers to ensure the output dimension size matches the input dimension size – the only exception to this rule is if I want to *purposely* reduce spatial dimensions via convolution. Applying zero-padding when *stacking* multiple CONV layers on top of each other has also demonstrated to increase classification accuracy in practice. As we'll see later in this book, libraries such as Keras can automatically compute zero-padding for you, making it even easier to build CNN architectures.

A second personal recommendation is to use POOL layers (rather than CONV layers) reduce the spatial dimensions of your input, at least until you become more experienced constructing your own CNN architectures. Once you reach that point, you should start experimenting with using CONV layers to reduce spatial input size and try removing max pooling layers from your architecture.

Most commonly, you'll see max pooling applied over a 2×2 receptive field size and a stride of $S = 2$. You might also see a 3×3 receptive field early in the network architecture to help reduce image size. It is **highly uncommon** to see receptive fields larger than three since these operations are very destructive to their inputs.

Batch normalization is an expensive operation which can *double* or *triple* the amount of time it takes to train your CNN; however, I recommend using BN in ***nearly all situations***. While BN does indeed slow down the training time, it also tends to "stabilize" training, making it easier to tune other hyperparameters (there are some exceptions, of course – I detail a few of these "exception architectures" inside the *ImageNet Bundle*).

I also place the batch normalization *after* the activation, as has become commonplace in the deep learning community even though it goes against the original Ioffe and Szegedy paper [123].

Inserting BN into the common layer architectures above, they become:

- `INPUT => CONV => RELU => BN => FC`
- `INPUT => [CONV => RELU => BN => POOL] * 2 => FC => RELU => BN => FC`
- `INPUT => [CONV => RELU => BN => CONV => RELU => BN => POOL] * 3 => [FC => RELU => BN] * 2 => FC`

You *do not* apply batch normalization before the softmax classifier as at this point we assume our network has learned its discriminative features earlier in the architecture.

Dropout (DO) is typically applied in between FC layers with a dropout probability of 50% – you should consider applying dropout in nearly *every* architecture you build. While not always performed, I also like to include dropout layers (with a very small probability, 10-25%) between POOL and CONV layers. Due to the local connectivity of CONV layers, dropout is less effective here,

but I've often found it helpful when battling overfitting.

By keeping these rules of thumb in mind, you'll be able to reduce your headaches when constructing CNN architectures since your `CONV` layers will preserve input sizes while the `POOL` layers take care of reducing spatial dimensions of the volumes, eventually leading to `FC` layers and the final output classifications.

Once you master this "traditional" method of building Convolutional Neural Networks, you should then start exploring leaving max pooling operations out *entirely* and using *just* `CONV` layers to reduce spatial dimensions, eventually leading to *average pooling* rather than an `FC` layer – these types of more advanced architecture techniques are covered inside the *ImageNet Bundle*.

11.4 Are CNNs Invariant to Translation, Rotation, and Scaling?

A common question I get asked is:

> *"Are Convolutional Neural Networks invariant to changes in translation, rotation, and scaling? Is that why they are such powerful image classifiers?"*

To answer this question, we first need to discriminate between the *individual filters* in the network along with the *final trained network*. Individual filters in a CNN are *not* invariant to changes in how an image is rotated – we demonstrate this in Chapter 12 of the *ImageNet Bundle* where we use features extracted from a CNN to determine how an image is oriented.

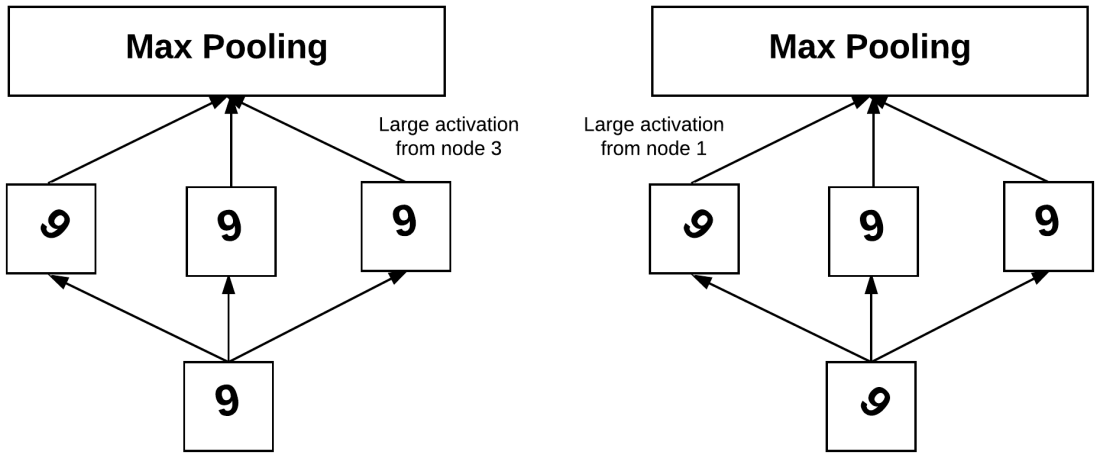

Figure 11.12: CNN as a whole learns filters that will fire when a pattern is presented at a particular orientation. On the *left* the, the digit 9 has been rotated $\approx 10^{\circ}$. This rotation is similar to node three which has learned what the digit 9 looks like when rotated in this manner. This node will have a higher activation than the other two nodes – the max pooling operation will detect this. On the *right* we have a second example, only this time the 9 has been rotated $\approx -45^{\circ}$, causing the first node to have the highest activation (Figure heavily inspired by Goodfellow et al. [10]).

However, a CNN *as a whole* can *learn* filters that fire when a pattern is presented at a particular orientation. For example, consider Figure 11.12, adapted and inspired from *Deep Learning* by Goodfellow et al. [10].

Here we see the digit *"9"* (bottom) presented to the CNN along with a set of filters the CNN has learned (middle). Since there is a filter inside the CNN that has "learned" what a *"9"* looks like, rotated by 10 degrees, it fires and emits a strong activation. This large activation is captured during the pooling stage and ultimately reported as the final classification.

The same is true for the second example (Figure 11.12, *left*). Here we see the *"9"* rotated by -45 degrees, and since there is a filter in the CNN that has learned what a *"9"* looks like when it is rotated by -45 degrees, the neuron activates and fires. Again, these filters themselves are *not* rotation invariant – it's just that the CNN has learned what a *"9"* looks like under *small rotations* that exist in the training set.

Unless your training data includes digits that are rotated across the full 360-degree spectrum, your CNN is *not* truly rotation invariant (again, this point is demonstrated in Chapter 12 of the *ImageNet Bundle*).

The same can be said about scaling – the filters themselves are *not* scale invariant, but it is highly likely that your CNN has learned a set of filters that *fire when patterns exist at varying scales*. We can also "help" our CNNs to be scale invariant by presenting our example image to them at testing time under varying scales and crops, then averaging the results together (see Chapter 10 of the *Practitioner Bundle* for more details on crop averaging to increase classification accuracy).

Translation invariance; however, is something that a CNN excels at. Keep in mind that a filter slides from left-to-right and top-to-bottom across an input, and will activate when it comes across a particular edge-like region, corner, or color blob. During the pooling operation, this large response is found and thus "beats" all its neighbors by having a larger activation. Therefore, CNNs can be seen as "not caring" exactly where an activation fires, simply that it *does fire* – and, in this way, we naturally handle translation inside a CNN.

11.5 Summary

In this chapter we took a tour of Convolutional Neural Network concepts (CNNs). We started by discussing what *convolution* and *cross-correlation* are and how the terms are used interchangeably in the deep learning literature.

To understand convolution at a more intimate level, we implemented it by hand using Python and OpenCV. However, traditional image processing operations require us to hand-define our kernels and are *specific* to a given image processing task (e.x., smoothing, edge detection, etc.). Using deep learning we can instead *learn* these types of filters which are then stacked on top of each other to automatically discover high-level concepts. We call this stacking and learning of higher-level features based on lower-level inputs the *compositionality* of Convolutional Neural Networks.

CNNs are built by stacking a sequence of layers where each layer is responsible for a given task. `CONV` layers will learn a set of K convolutional filters, each of which are size $F \times F$ pixels. We then apply activation layers on top of the `CONV` layers to obtain a nonlinear transformation. `POOL` layers help reduce the spatial dimensions of the input volume as it flows through the network.

Once the input volume is sufficiently small, we can apply `FC` layers which are our traditional dot product layers from Chapter 12, eventually feeding into a softmax classifier for our final output predictions.

Batch normalization layers are used to standardize inputs to a `CONV` or activation layer by computing the mean and standard deviation across a mini-batch. A dropout layer can then be applied to randomly disconnect nodes from a given input to an output, helping to reduce overfitting.

Finally, we wrapped up the chapter by reviewing common CNN architectures that you can use to implement your own networks. In our next chapter, we'll implement your first CNN in Keras, ShallowNet, based on the layer patterns we mentioned above. Future chapters will discuss deeper network architectures such as the seminal LeNet architecture [19] and variants of the VGGNet architecture [95].

12. Training Your First CNN

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

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

12.1 Keras Configurations and Converting Images to Arrays

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

12.1.1 Understanding the keras.json Configuration File

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

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

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

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

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

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

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

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

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

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

12.1.2 The Image to Array Preprocessor

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

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

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

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

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

Line 2 imports the `img_to_array` function from Keras.

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

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

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

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

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

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

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

```
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

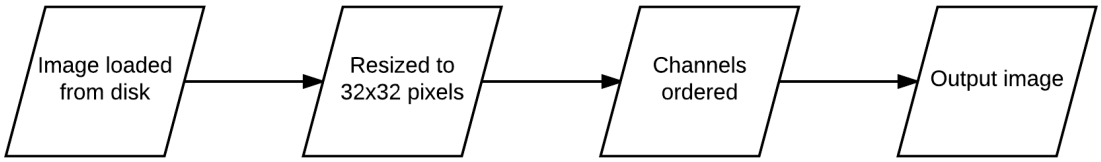

Figure 12.1: An example image pre-processing pipeline that (1) loads an image from disk, (2) resizes it to 32×32 pixels, (3) orders the channel dimensions, and (4) outputs the image.

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:

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

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

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

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

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

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

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

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

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

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

On **Line 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:

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

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

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

12.2.2 ShallowNet on Animals

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

From there, we can get to work:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

After training our network, we can evaluate its performance:

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

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

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

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

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

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

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

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

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

avg / total       0.71      0.69      0.68       750
```

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

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

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

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

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

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

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

12.2.3 ShallowNet on CIFAR-10

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

         avg / total       0.60      0.59      0.59     10000
```

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

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

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

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

12.3 Summary

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

When applied to CIFAR-10, ShallowNet reached 60% accuracy, 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.

13. Saving and Loading Your Models

In our last chapter, you learned how to train your first Convolutional Neural Network using the Keras library. However, you might have noticed that each time you wanted to evaluate your network or test it on a set of images, you first needed to train it *before* you could do any type of evaluation. This requirement can be quite the nuisance.

We are only working with a shallow network on a small dataset which can be trained relatively quickly, but what if our network was deep and we needed to train it on a much larger dataset, thus taking many hours or even days to train? Would we have to invest this amount of time and resources to train our network *each and every time?* Or is there a way to *save* our model to disk after training is complete and then simply load it from disk when we want to classify new images?

You bet there's a way. The process of saving and loading a trained model is called **model serialization** and is the primary topic of this chapter.

13.1 Serializing a Model to Disk

Using the Keras library, model serialization is as simple as calling `model.save` on a trained model and then loading it via the `load_model` function. In the first part of this chapter, we'll modify our ShallowNet training script from the last chapter to serialize the network after it's been trained on the Animals dataset. We'll then create a second Python script that demonstrates how to load our serialized model from disk.

Let's get started with the training part – open up a new file, name it `shallownet_train.py`, and insert the following code:

```
# 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
```

```
8  from pyimagesearch.nn.conv import ShallowNet
9  from keras.optimizers import SGD
10 from imutils import paths
11 import matplotlib.pyplot as plt
12 import numpy as np
13 import argparse
```

Lines 2-13 import our required Python packages. Much of the code in this example is identical to `shallownet_animals.py` from Chapter 12. We'll review the entire file, for the sake of completeness, and I'll be sure to call out the important changes made to accomplish model serialization, but for a detailed review of how to train ShallowNet on the Animals dataset, please refer Section 12.2.1.

Next, let's parse our command line arguments:

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

Our previous script only required a *single* switch, `--dataset`, which is the path to the input Animals dataset. However, as you can see, we've added another switch here – `--model` which is the path to where we would like to *save network after training is complete*.

We can now grab the paths to the images in our `--dataset`, initialize our preprocessors, and load our image dataset from disk:

```
23 # grab the list of images that we'll be describing
24 print("[INFO] loading images...")
25 imagePaths = list(paths.list_images(args["dataset"]))
26
27 # initialize the image preprocessors
28 sp = SimplePreprocessor(32, 32)
29 iap = ImageToArrayPreprocessor()
30
31 # load the dataset from disk then scale the raw pixel intensities
32 # to the range [0, 1]
33 sdl = SimpleDatasetLoader(preprocessors=[sp, iap])
34 (data, labels) = sdl.load(imagePaths, verbose=500)
35 data = data.astype("float") / 255.0
```

The next step is to partition our data into training and testing splits, along with encoding our labels as vectors:

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

```
trainY = LabelBinarizer().fit_transform(trainY)
testY = LabelBinarizer().fit_transform(testY)
```

Training ShallowNet is handled via the code block below:

```
# initialize the optimizer and model
print("[INFO] compiling model...")
opt = SGD(lr=0.005)
model = ShallowNet.build(width=32, height=32, depth=3, classes=3)
model.compile(loss="categorical_crossentropy", optimizer=opt,
	metrics=["accuracy"])

# train the network
print("[INFO] training network...")
H = model.fit(trainX, trainY, validation_data=(testX, testY),
	batch_size=32, epochs=100, verbose=1)
```

Now that our network is trained, we need to save it to disk. This process is as simple as calling `model.save` and supplying the path to where our output network should be saved to disk:

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

The `.save` method takes the weights and state of the optimizer and serializes them to disk in HDF5 format. As we'll see in the next section, loading these weights from disk is just as easy as saving them.

From here we evaluate our network:

```
# 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=["cat", "dog", "panda"]))
```

As well as plot our loss and accuracy:

```
# plot the training loss and accuracy
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, 100), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, 100), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, 100), H.history["acc"], label="train_acc")
plt.plot(np.arange(0, 100), H.history["val_acc"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
plt.show()
```

To run our script, simply execute the following command:

```
$ python shallownet_train.py --dataset ../datasets/animals \
    --model shallownet_weights.hdf5
```

After the network has finished training, list the contents of your directory:

```
$ ls
shallownet_load.py  shallownet_train.py  shallownet_weights.hdf5
```

And you will see a file named `shallownet_weights.hdf5` – this file is our serialized network. The next step is to take this saved network and load it from disk.

13.2 Loading a Pre-trained Model from Disk

Now that we've trained our model and serialized it, we need to load it from disk. As a practical application of model serialization, I'll be demonstrating how to classify *individual images* from the Animals dataset and then display the classified images to our screen.

Open a new file, name it `shallownet_load.py`, and we'll get our hands dirty:

```
1  # import the necessary packages
2  from pyimagesearch.preprocessing import ImageToArrayPreprocessor
3  from pyimagesearch.preprocessing import SimplePreprocessor
4  from pyimagesearch.datasets import SimpleDatasetLoader
5  from keras.models import load_model
6  from imutils import paths
7  import numpy as np
8  import argparse
9  import cv2
```

We start off by importing our required Python packages. **Lines 2-4** import the classes uses to construct our standard pipeline of resizing an image to a fixed size, converting it to a Keras compatible array, and then using these preprocessors to load an entire image dataset into memory.

The actual function used to load our trained model from disk is `load_model` on **Line 5**. This function is responsible for accepting the path to our trained network (an HDF5 file), decoding the weights and optimizer inside the HDF5 file, and setting the weights inside our architecture so we can (1) continue training or (2) use the network to classify new images.

We'll import our OpenCV bindings on **Line 9** as well so we can draw the classification label on our images and display them to our screen.

Next, let's parse our command line arguments:

```
11 # construct the argument parse and parse the arguments
12 ap = argparse.ArgumentParser()
13 ap.add_argument("-d", "--dataset", required=True,
14     help="path to input dataset")
15 ap.add_argument("-m", "--model", required=True,
16     help="path to pre-trained model")
17 args = vars(ap.parse_args())
18
19 # initialize the class labels
20 classLabels = ["cat", "dog", "panda"]
```

Just like in `shallownet_save.py`, we'll need two command line arguments:

1. `--dataset`: The path to the directory that contains images that we wish to classify (in this case, the Animals dataset).
2. `--model`: The path to the *trained network* serialized on disk.

Line 20 then initializes a list of class labels for the Animals dataset.

Our next code block handles randomly sampling ten image paths from the Animals dataset for classification:

```
22 # grab the list of images in the dataset then randomly sample
23 # indexes into the image paths list
24 print("[INFO] sampling images...")
25 imagePaths = np.array(list(paths.list_images(args["dataset"])))
26 idxs = np.random.randint(0, len(imagePaths), size=(10,))
27 imagePaths = imagePaths[idxs]
```

Each of these ten images will need to be preprocessed, so let's initialize our preprocessors and load the ten images from disk:

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

Notice how we are preprocessing our images in the *exact same manner* in which we preprocessed our images during training. Failing to do this procedure can lead to incorrect classifications since the network will be presented with patterns it cannot recognize. Always take special care to ensure your *testing images* were preprocessed in the same way as your *training images*.

Next, let's load our saved network from disk:

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

Loading our serialized network is as simple as calling `load_model` and supplying the path to model's HDF5 file residing on disk.

Once the model is loaded, we can make predictions on our ten images:

```
43 # make predictions on the images
44 print("[INFO] predicting...")
45 preds = model.predict(data, batch_size=32).argmax(axis=1)
```

Keep in mind that the `.predict` method of `model` will return a *list of probabilities* for every image in `data` – one probability for each class label, respectively. Taking the `argmax` on `axis=1` finds the index of the class label with the *largest probability* for each image.

Now that we have our predictions, let's visualize the results:

```
47 # loop over the sample images
48 for (i, imagePath) in enumerate(imagePaths):
49     # load the example image, draw the prediction, and display it
50     # to our screen
51     image = cv2.imread(imagePath)
52     cv2.putText(image, "Label: {}".format(classLabels[preds[i]]),
53         (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)
54     cv2.imshow("Image", image)
55     cv2.waitKey(0)
```

On **Line 48** we start looping over our ten randomly sampled image paths. For each image, we load it from disk (**Line 51**) and draw the class label prediction on the image itself (**Lines 52 and 53**). The output image is then displayed to our screen on **Lines 54 and 55**.

To give `shallownet_load.py` a try, execute the following command:

```
$ python shallownet_load.py --dataset ../datasets/animals \
    --model shallownet_weights.hdf5
[INFO] sampling images...
[INFO] loading pre-trained network...
[INFO] predicting...
```

Based on the output, you can see that our images have been sampled, the pre-trained ShallowNet weights have been loaded from disk, and that ShallowNet has made predictions on our images. I have included a sample of predictions from the ShallowNet drawn on the images themselves in Figure 13.1.

Figure 13.1: A sample of images correctly classified by our ShallowNet CNN.

Keep in mind that ShallowNet is obtaining $\approx 70\%$ classification accuracy on the Animals dataset, meaning that nearly one in every three example images will be classified incorrectly. Furthermore, based on the `classification_report` from Section 12.2.2, we know that the network still struggles to consistently discriminate between dogs and cats. As we continue our journey applying deep learning to computer vision classification tasks, we'll look at methods to help us boost our classification accuracy.

13.3 Summary

In this chapter we learned how to:

1. Train a network.
2. Serialize the network weights and optimizer state to disk.
3. Load the trained network and classify images.

Later in Chapter 18 we'll discover how we can save our model's weights to disk after *every epoch*, allowing us to "checkpoint" our network and choose the best performing one. Saving model weights during the actual training process also enables us to *restart training from a specific point* if our network starts exhibiting signs of overfitting. The process of stopping training, tweaking parameters, and then restarting training again is covered in-depth inside the *Practitioner Bundle* and *ImageNet Bundle*.

14. LeNet: Recognizing Handwritten Digits

The LeNet architecture is a seminal work in the deep learning community, first introduced by LeCun et al. in their 1998 paper, *Gradient-Based Learning Applied to Document Recognition* [19]. As the name of the paper suggests, the authors' motivation behind implementing LeNet was primarily for Optical Character Recognition (OCR).

The LeNet architecture is *straightforward* and *small* (in terms of memory footprint), making it *perfect* for teaching the basics of CNNs.

In this chapter, we'll seek to replicate experiments similar to LeCun's in their 1998 paper. We'll start by reviewing the LeNet architecture and then implement the network using Keras. Finally, we'll evaluate LeNet on the MNIST dataset for handwritten digit recognition.

14.1 The LeNet Architecture

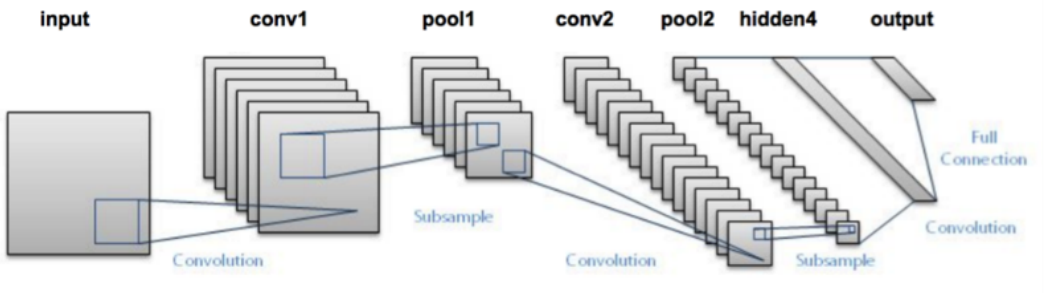

Figure 14.1: The LeNet architecture consists of two series of `CONV => TANH => POOL` layer sets followed by a fully-connected layer and softmax output. *Photo Credit: http://pyimg.co/ihjsx*

Now that we have explored the building blocks of Convolutional Neural Networks in Chapter 12 using ShallowNet, we are ready to take the next step and discuss LeNet. The LeNet architecture

Layer Type	Output Size	Filter Size / Stride
INPUT IMAGE	$28 \times 28 \times 1$	
CONV	$28 \times 28 \times 20$	$5 \times 5, K = 20$
ACT	$28 \times 28 \times 20$	
POOL	$14 \times 14 \times 20$	2×2
CONV	$14 \times 14 \times 50$	$5 \times 5, K = 50$
ACT	$14 \times 14 \times 50$	
POOL	$7 \times 7 \times 50$	2×2
FC	500	
ACT	500	
FC	10	
SOFTMAX	10	

Table 14.1: A table summary of the LeNet architecture. Output volume sizes are included for each layer, along with convolutional filter size/pool size when relevant.

(Figure 14.1) is an excellent first "real-world" network. The network is small and easy to understand — yet large enough to provide interesting results.

Furthermore, the combination of LeNet + MNIST is able to be easily run on the CPU, making it easy for beginners to take their first step in deep learning and CNNs. In many ways, LeNet + MNIST is the *"Hello, World"* equivalent of deep learning applied to image classification. The LeNet architecture consists of the following layers, using a pattern of `CONV => ACT => POOL` from Section 11.3:

```
INPUT => CONV => TANH => POOL => CONV => TANH => POOL =>
     FC => TANH => FC
```

Notice how the LeNet architecture uses the *tanh* activation function rather than the more popular *ReLU*. Back in 1998 the ReLU had not been used in the context of deep learning — it was more common to use *tanh* or *sigmoid* as an activation function. When implementing LeNet today, it's common to swap out `TANH` for `RELU` — we'll follow this same guideline and use ReLU as our activation function later in this chapter.

Table 14.1 summarizes the parameters for the LeNet architecture. Our *input layer* takes an input image with 28 rows, 28 columns, and a single channel (grayscale) for depth (i.e., the dimensions of the images inside the MNIST dataset). We then learn 20 filters, each of which are 5×5. The `CONV` layer is followed by a ReLU activation followed by max pooling with a 2×2 size and 2×2 stride.

The next block of the architecture follows the same pattern, this time learning 50 5×5 filters. It's common to see the number of `CONV` layers *increase* in deeper layers of the network as the actual spatial input dimensions *decrease*.

We then have two `FC` layers. The first `FC` contains 500 hidden nodes followed by a ReLU activation. The final `FC` layer controls the number of output class labels (0-9; one for each of the possible ten digits). Finally, we apply a softmax activation to obtain the class probabilities.

14.2 Implementing LeNet

Given Table 14.1 above, we are now ready to implement the seminal LeNet architecture using the Keras library. Begin by adding a new file named `lenet.py` inside the `pyimagesearch.nn.conv` sub-module — this file will store our actual LeNet implementation:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- nn
|    |    |--- __init__.py
...
|    |    |--- conv
|    |    |    |--- __init__.py
|    |    |    |--- lenet.py
|    |    |    |--- shallownet.py
```

From there, open up `lenet.py`, and we can start coding:

```
1  # import the necessary packages
2  from keras.models import Sequential
3  from keras.layers.convolutional import Conv2D
4  from keras.layers.convolutional import MaxPooling2D
5  from keras.layers.core import Activation
6  from keras.layers.core import Flatten
7  from keras.layers.core import Dense
8  from keras import backend as K
```

Lines 2-8 handle importing our required Python packages — these imports are exactly the same as the ShallowNet implementation from Chapter 12 and form the essential set of required imports when building (nearly) any CNN using Keras.

We then define the `build` method of `LeNet` below, used to actually construct the network architecture:

```
10 class LeNet:
11     @staticmethod
12     def build(width, height, depth, classes):
13         # initialize the model
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)
```

The build method requires four parameters:

1. The *width* of the input image.
2. The *height* of the input image.
3. The *number of channels* (depth) of the image.
4. The number *class labels* in the classification task.

The `Sequential` class, the building block of sequential networks sequentially stack one layer on top of the other is initialized on **Line 14**. We then initialize the `inputShape` as if using "channels last" ordering. In the case that our Keras configuration is set to use "channels first" ordering, we update the `inputShape` on **Lines 18 and 19**.

The first set of `CONV => RELU => POOL` layers are defined below:

```
            # first set of CONV => RELU => POOL layers
            model.add(Conv2D(20, (5, 5), padding="same",
                input_shape=inputShape))
            model.add(Activation("relu"))
            model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
```

Our `CONV` layer will learn 20 filters, each of size 5×5. We then apply a ReLU activation function followed by a 2×2 pooling with a 2×2 stride, thereby decreasing the input volume size by 75%.

Another set of `CONV => RELU => POOL` layers are then applied, this time learning 50 filters rather than 20:

```
            # second set of CONV => RELU => POOL layers
            model.add(Conv2D(50, (5, 5), padding="same"))
            model.add(Activation("relu"))
            model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
```

The input volume can then be flattened and a fully-connected layer with 500 nodes can be applied:

```
            # first (and only) set of FC => RELU layers
            model.add(Flatten())
            model.add(Dense(500))
            model.add(Activation("relu"))
```

Followed by the final softmax classifier:

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

            # return the constructed network architecture
            return model
```

Now that we have coded up the LeNet architecture, we can move on to applying it to the MNIST dataset.

14.3 LeNet on MNIST

Our next step is to create a driver script that is responsible for:

1. Loading the MNIST dataset from disk.
2. Instantiating the LeNet architecture.
3. Training LeNet.
4. Evaluating network performance.

To train and evaluate LeNet on MNIST, create a new file named `lenet_mnist.py`, and we can get started:

```
# import the necessary packages
from pyimagesearch.nn.conv import LeNet
```

```
3  from keras.optimizers import SGD
4  from sklearn.preprocessing import LabelBinarizer
5  from sklearn.model_selection import train_test_split
6  from sklearn.metrics import classification_report
7  from sklearn import datasets
8  from keras import backend as K
9  import matplotlib.pyplot as plt
10 import numpy as np
```

At this point, our Python imports should start to feel pretty standard with a noticeable pattern appearing. In the vast majority of examples in this book, we'll have to import:

1. A *network architecture* that we are going to train.
2. An *optimizer* to train the network (in this case, SGD).
3. A (set of) convenience function(s) used to construct the training and testing splits of a given dataset.
4. A function to compute a classification report so we can evaluate our classifier's performance.

Again, nearly all examples in this book will follow this import pattern, along with a few extra classes here and there to facilitate certain tasks (such as preprocessing images). The MNIST dataset has already been preprocessed so we can simply load via the following function call:

```
12 # grab the MNIST dataset (if this is your first time using this
13 # dataset then the 55MB download may take a minute)
14 print("[INFO] accessing MNIST...")
15 dataset = datasets.fetch_mldata("MNIST Original")
16 data = dataset.data
```

Line 15 loads the MNIST dataset from disk. If this is your first time calling the `fetch_mldata` function with the `"MNIST Original"` string, then the MNIST dataset will need to be downloaded from the mldata.org dataset repository. The MNIST dataset is serialized into a single 55MB file, so depending on your internet connection, this download may take anywhere from a couple of seconds to a couple of minutes.

It's important to note that each MNIST sample inside `data` is represented by a 784-d vector (i.e., the raw pixel intensities) of a 28×28 grayscale mage. Therefore, we need to reshape the `data` matrix depending on whether we are using "channels first" or "channels last" ordering:

```
18 # if we are using "channels first" ordering, then reshape the
19 # design matrix such that the matrix is:
20 # num_samples x depth x rows x columns
21 if K.image_data_format() == "channels_first":
22     data = data.reshape(data.shape[0], 1, 28, 28)
23
24 # otherwise, we are using "channels last" ordering, so the design
25 # matrix shape should be: num_samples x rows x columns x depth
26 else:
27     data = data.reshape(data.shape[0], 28, 28, 1)
```

If we are performing "channels first" ordering (**Lines 21 and 22**), then the `data` matrix is reshaped such that the number of samples is the first entry in the matrix, the single channel as the second entry, followed by the number of rows and columns (28 and 28 respectively). Otherwise, we assume we are using "channels last" ordering in which case the matrix is reshaped as number of

samples first, number of rows, number of columns, and finally the number of channels (**Lines 26 and 27**).

Now that our data matrix is properly shaped, we can perform a training and testing split, taking care to scale the image pixel intensities to the range $[0, 1]$ first:

```
29 # scale the input data to the range [0, 1] and perform a train/test
30 # split
31 (trainX, testX, trainY, testY) = train_test_split(data / 255.0,
32      dataset.target.astype("int"), test_size=0.25, random_state=42)
33
34 # convert the labels from integers to vectors
35 le = LabelBinarizer()
36 trainY = le.fit_transform(trainY)
37 testY = le.transform(testY)
```

After splitting the data, we also encode our class labels as one-hot vectors rather than single integer values. For example, if the class label for a given sample was 3, then the output of one-hot encoding the label would be:

```
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
```

Notice how all entries in the vector are zero *except* for the fourth index which is now set to one (keep in mind that the digit 0 is the first index, hence why three is the *fourth* index).

The stage is now set to train LeNet on MNIST:

```
39 # initialize the optimizer and model
40 print("[INFO] compiling model...")
41 opt = SGD(lr=0.01)
42 model = LeNet.build(width=28, height=28, depth=1, classes=10)
43 model.compile(loss="categorical_crossentropy", optimizer=opt,
44      metrics=["accuracy"])
45
46 # train the network
47 print("[INFO] training network...")
48 H = model.fit(trainX, trainY, validation_data=(testX, testY),
49      batch_size=128, epochs=20, verbose=1)
```

Line 41 initializes our `SGD` optimizer with a learning rate of `0.01`. `LeNet` itself is instantiated on **Line 42**, indicating that all input images in our dataset will be 28 pixels wide, 28 pixels tall, and have a depth of 1. Given that there are ten classes in the MNIST dataset (one for each of the digits, $0-9$), we set `classes=10`.

Lines 43 and 44 compile the model using cross-entropy loss as our loss function. **Line 48 and 49** trains LeNet on MNIST for a total of 20 epochs using a mini-batch size of 128.

Finally, we can evaluate the performance on our network as well as plot the loss and accuracy over time in the final code block below:

```
51 # evaluate the network
52 print("[INFO] evaluating network...")
53 predictions = model.predict(testX, batch_size=128)
54 print(classification_report(testY.argmax(axis=1),
55      predictions.argmax(axis=1),
56      target_names=[str(x) for x in le.classes_]))
57
```

```
58 # plot the training loss and accuracy
59 plt.style.use("ggplot")
60 plt.figure()
61 plt.plot(np.arange(0, 20), H.history["loss"], label="train_loss")
62 plt.plot(np.arange(0, 20), H.history["val_loss"], label="val_loss")
63 plt.plot(np.arange(0, 20), H.history["acc"], label="train_acc")
64 plt.plot(np.arange(0, 20), H.history["val_acc"], label="val_acc")
65 plt.title("Training Loss and Accuracy")
66 plt.xlabel("Epoch #")
67 plt.ylabel("Loss/Accuracy")
68 plt.legend()
69 plt.show()
```

I mentioned this fact before in Section 12.2.2 when evaluating ShallowNet, but make sure you understand what **Line 53** is doing when `model.predict` is called. For each sample in `testX`, batch sizes of 128 are constructed and then passed through the network for classification. After all testing data points have been classified, the `predictions` variable is returned.

The `predictions` variable is actually a NumPy array with the shape `(len(testX), 10)` implying that we now have the 10 probabilities associated with *each* class label for *every* data point in `testX`. Taking `predictions.argmax(axis=1)` in `classification_report` on **Lines 54-56** finds the index of the label with the *largest probability* (i.e., the final output classification). Given the final classification from the network, we can compare our *predicted* class labels to the *ground-truth* labels.

To execute our script, just issue the following command:

```
$ python lenet_mnist.py
```

The MNIST dataset should then be downloaded and/or loaded from disk and training should commence:

```
[INFO] accessing MNIST...
[INFO] compiling model...
[INFO] training network...
Train on 52500 samples, validate on 17500 samples
Epoch 1/20
3s - loss: 1.0970 - acc: 0.6976 - val_loss: 0.5348 - val_acc: 0.8228
...
Epoch 20/20
3s - loss: 0.0411 - acc: 0.9877 - val_loss: 0.0576 - val_acc: 0.9837
[INFO] evaluating network...
              precision    recall  f1-score   support

           0       0.99      0.99      0.99      1677
           1       0.99      0.99      0.99      1935
           2       0.99      0.98      0.99      1767
           3       0.99      0.97      0.98      1766
           4       1.00      0.98      0.99      1691
           5       0.99      0.98      0.98      1653
           6       0.99      0.99      0.99      1754
           7       0.98      0.99      0.99      1846
           8       0.94      0.99      0.97      1702
           9       0.98      0.98      0.98      1709
```

```
avg / total       0.98      0.98      0.98     17500
```

Using my Titan X GPU I was obtaining three-second epochs. Using *just* the CPU, the number of seconds per epoch jumped to thirty. After training completes, we can see that LeNet is obtaining **98%** classification accuracy, a *huge* increase from 92% when using standard feedforward neural networks in Chapter 10.

Furthermore, looking at our loss and accuracy plot over time in Figure 14.2 demonstrates that our network is behaving quite well. After only five epochs LeNet is *already* reaching $\approx 96\%$ classification accuracy. Loss on both the training and validation data continues to fall with only a handful of minor "spikes" due to our learning rate staying constant and not decaying (a concept we'll cover later in Chapter 16). At the end of the twentieth epoch, we are reaching 98% accuracy on our testing set.

Figure 14.2: Training LeNet on MNIST. After only twenty epochs we are obtaining 98% classification accuracy.

This plot demonstrating the loss and accuracy of LeNet on MNIST is arguably the *quintessential* graph we are looking for: the training and validation loss and accuracy mimic each other (nearly) exactly with no signs of overfitting. As we'll see, it's often *very hard* to obtain this type of training plot that behaves so nicely, indicating that our network is learning the underlying patterns *without* overfitting.

There is also the problem that the MNIST dataset is *heavily preprocessed* and not representative of image classification problems we'll encounter in the real-world. Researchers tend to use the MNIST dataset as a benchmark to evaluate new classification algorithms. If their methods cannot obtain $> 95\%$ classification accuracy, then there is either a flaw in (1) the logic of the algorithm or

(2) the implementation itself.

Nonetheless, applying LeNet to MNIST is an excellent way to get your first taste at applying deep learning to image classification problems and mimicking the results of the seminal LeCun et al. paper.

14.4 Summary

In this chapter, we explored the LeNet architecture, introduced by LeCun et al. in their 1998 paper, *Gradient-Based Learning Applied to Document Recognition* [19]. LeNet is a seminal work in the deep learning literature — it thoroughly demonstrated how neural networks could be trained to recognize objects in images in an end-to-end manner (i.e., no feature extraction had to take place, the network was able to learn patterns from the images themselves).

While seminal, LeNet by today's standards is still considered a "shallow" network. With only four trainable layers (two `CONV` layers and two `FC` layers), the depth of LeNet pales in comparison to the depth of current state-of-the-art architectures such as VGG (16 and 19 layers) and ResNet (100+ layers).

In our next chapter, we'll discuss a variation of the VGGNet architecture which I call *"MiniVGGNet"*. This variation of the architecture uses the exact same guiding principles as Simonyan and Zisserman's work [95], but reduces the depth, allowing us to train the network on smaller datasets. For a *full* implementation of the VGGNet architecture, you'll want to refer to Chapter 6 of the *ImageNet Bundle* where we train VGGNet from scratch on ImageNet.

15. MiniVGGNet: Going Deeper with CNNs

In our previous chapter we discussed LeNet, a seminal Convolutional Neural Network in the deep learning and computer vision literature. VGGNet, (sometimes referred to as simply VGG), was first introduced by Simonyan and Zisserman in their 2014 paper, *Very Deep Learning Convolutional Neural Networks for Large-Scale Image Recognition* [95]. The primary contribution of their work was demonstrating that an architecture with very small (3×3) filters can be trained to increasingly higher depths (16-19 layers) and obtain state-of-the-art classification on the challenging ImageNet classification challenge.

Previously, network architectures in the deep learning literature used a mix of filter sizes:

The first layer of the CNN usually includes filter sizes somewhere between 7×7 [94] and 11×11 [128]. From there, filter sizes progressively reduced to 5×5. Finally, only the deepest layers of the network used 3×3 filters.

VGGNet is unique in that it uses 3×3 kernels **throughout the entire architecture**. The use of these small kernels is arguably what helps VGGNet *generalize* to classification problems outside what the network was originally trained on (we'll see this inside the *Practitioner Bundle* and *ImageNet Bundle* when we discuss transfer learning).

Any time you see a network architecture that consists *entirely* of 3×3 filters, you can rest assured that it was inspired by VGGNet. Reviewing the *entire* 16 and 19 layer variants of VGGNet is too advanced for this introduction to Convolutional Neural Networks – for a detailed review of VGG16 and VGG19, please refer to the Chapter 11 of the *ImageNet Bundle*.

Instead, we are going to review the VGG family of networks and define what characteristics a CNN must exhibit to fit into this family. From there we'll implement a smaller version of VGGNet called *MiniVGGNet* that can easily be trained on your system. This implementation will also demonstrate how to use two important layers we discussed in Chapter 11 – *batch normalization* (BN) and *dropout*.

15.1 The VGG Family of Networks

The VGG family of Convolutional Neural Networks can be characterized by two key components:

1. All `CONV` layers in the network using *only* 3×3 filters.

2. Stacking *multiple* `CONV => RELU` layer sets (where the number of consecutive `CONV => RELU` layers normally *increases* the *deeper* we go) before applying a `POOL` operation.

In this section, we are going to discuss a variant of the VGGNet architecture which I call "MiniVGGNet" due to the fact that the network is substantially more shallow than its big brother. For a detailed review and implementation of the original VGG architecture proposed by Simonyan and Zisserman, along with a demonstration on how to train the network on the ImageNet dataset, please refer to Chapter 11 of the *ImageNet Bundle*.

15.1.1 The (Mini) VGGNet Architecture

In both ShallowNet and LeNet we have applied a series of `CONV => RELU => POOL` layers. However, in VGGNet, we stack *multiple* `CONV => RELU` layers prior to applying a single `POOL` layer. Doing this allows the network to learn more rich features from the `CONV` layers prior to downsampling the spatial input size via the `POOL` operation.

Overall, MiniVGGNet consists of *two sets* of `CONV => RELU => CONV => RELU => POOL` layers, followed by a set of `FC => RELU => FC => SOFTMAX` layers. The first two `CONV` layers will learn 32 filters, each of size 3×3. The second two `CONV` layers will learn 64 filters, again, each of size 3×3. Our `POOL` layers will perform max pooling over a 2×2 window with a 2×2 stride. We'll also be inserting batch normalization layers *after* the activations along with dropout layers (`DO`) after the `POOL` and `FC` layers.

The network architecture itself is detailed in Table 15.1, where the initial input image size is assumed to be $32 \times 32 \times 3$ as we'll be training MiniVGGNet on CIFAR-10 later in this chapter (and then comparing performance to ShallowNet).

Again, notice how the batch normalization and dropout layers are included in the network architecture based on my *"Rules of Thumb"* in Section 11.3.2. Applying batch normalization will help reduce the effects of overfitting and increase our classification accuracy on CIFAR-10.

15.2 Implementing MiniVGGNet

Given the description of MiniVGGNet in Table 15.1, we can now implement the network architecture using Keras. To get started, add a new file named `minivggnet.py` inside the `pyimagesearch.nn.conv` sub-module – there is where we will write our MiniVGGNet implementation:

```
--- pyimagesearch
|    |--- __init__.py
|    |--- nn
|    |    |--- __init__.py
...
|    |    |--- conv
|    |    |    |--- __init__.py
|    |    |    |--- lenet.py
|    |    |    |--- minivggnet.py
|    |    |    |--- shallownet.py
```

After creating the `minivggnet.py` file, open it up in your favorite code editor and we'll get to work:

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

Layer Type	**Output Size**	**Filter Size / Stride**
INPUT IMAGE	$32 \times 32 \times 3$	
CONV	$32 \times 32 \times 32$	$3 \times 3, K = 32$
ACT	$32 \times 32 \times 32$	
BN	$32 \times 32 \times 32$	
CONV	$32 \times 32 \times 32$	$3 \times 3, K = 32$
ACT	$32 \times 32 \times 32$	
BN	$32 \times 32 \times 32$	
POOL	$16 \times 16 \times 32$	2×2
DROPOUT	$16 \times 16 \times 32$	
CONV	$16 \times 16 \times 64$	$3 \times 3, K = 64$
ACT	$16 \times 16 \times 64$	
BN	$16 \times 16 \times 64$	
CONV	$16 \times 16 \times 64$	$3 \times 3, K = 64$
ACT	$16 \times 16 \times 64$	
BN	$16 \times 16 \times 64$	
POOL	$8 \times 8 \times 64$	2×2
DROPOUT	$8 \times 8 \times 64$	
FC	512	
ACT	512	
BN	512	
DROPOUT	512	
FC	10	
SOFTMAX	10	

Table 15.1: A table summary of the MiniVGGNet architecture. Output volume sizes are included for each layer, along with convolutional filter size/pool size when relevant. Notice how only 3×3 convolutions are applied.

```
5  from keras.layers.convolutional import MaxPooling2D
6  from keras.layers.core import Activation
7  from keras.layers.core import Flatten
8  from keras.layers.core import Dropout
9  from keras.layers.core import Dense
10 from keras import backend as K
```

Lines 2-10 import our required classes from the Keras library. Most of these imports you have already seen before, but I want to bring your attention to the `BatchNormalization` (**Line 3**) and `Dropout` (**Line 8**) – these classes will enable us to apply batch normalization and dropout to our network architecture.

Just like our implementations of both ShallowNet and LeNet, we'll define a `build` method that can be called to construct the architecture using a supplied `width`, `height`, `depth`, and number of `classes`:

```
12 class MiniVGGNet:
13     @staticmethod
14     def build(width, height, depth, classes):
15         # initialize the model along with the input shape to be
16         # "channels last" and the channels dimension itself
17         model = Sequential()
18         inputShape = (height, width, depth)
19         chanDim = -1
20
21         # if we are using "channels first", update the input shape
22         # and channels dimension
23         if K.image_data_format() == "channels_first":
24             inputShape = (depth, height, width)
25             chanDim = 1
```

Line 17 instantiates the `Sequential` class, the building block of sequential neural networks in Keras. We then initialize the `inputShape`, assuming we are using channels last ordering (**Line 18**).

Line 19 introduces a variable we haven't seen before, `chanDim`, *the index of the channel dimension*. Batch normalization operates over the channels, so in order to apply BN, we need to know which axis to normalize over. Setting `chanDim = -1` implies that the index of the channel dimension *last* in the input shape (i.e., channels last ordering). However, if we are using channels first ordering (**Lines 23-25**), we need need to update the `inputShape` and set `chanDim = 1`, since the channel dimension is now the first entry in the input shape.

The first layer block of MiniVGGNet is defined below:

```
27         # first CONV => RELU => CONV => RELU => POOL layer set
28         model.add(Conv2D(32, (3, 3), padding="same",
29             input_shape=inputShape))
30         model.add(Activation("relu"))
31         model.add(BatchNormalization(axis=chanDim))
32         model.add(Conv2D(32, (3, 3), padding="same"))
33         model.add(Activation("relu"))
34         model.add(BatchNormalization(axis=chanDim))
35         model.add(MaxPooling2D(pool_size=(2, 2)))
36         model.add(Dropout(0.25))
```

Here we can see our architecture consists of `(CONV => RELU => BN) * 2 => POOL => DO`. **Line 28** defines a `CONV` layer with 32 filters, each of which has a 3×3 filter size. We then apply a ReLU activation (**Line 30**) which is immediately fed into a `BatchNormalization` layer (**Line 31**) to zero-center the activations.

However, instead of applying a `POOL` layer to reduce the spatial dimensions of our input, we instead apply another set of `CONV => RELU => BN` – this allows our network to learn more rich features, a common practice when training deeper CNNs.

On **Line 35** we use `MaxPooling2D` with a size of 2×2. Since we do not *explicitly* set a stride, Keras *implicitly* assumes our stride to be equal to the max pooling size (which is 2×2).

We then apply `Dropout` on **Line 36** with a probability of $p = 0.25$, which this implies that a node from the `POOL` layer will randomly disconnect from the next layer with a probability of 25% during training. We apply dropout to help reduce the effects of overfitting. You can read more about dropout in Section 11.2.7. We then add the second layer block to MiniVGGNet below:

```
38             # second CONV => RELU => CONV => RELU => POOL layer set
39             model.add(Conv2D(64, (3, 3), padding="same"))
40             model.add(Activation("relu"))
41             model.add(BatchNormalization(axis=chanDim))
42             model.add(Conv2D(64, (3, 3), padding="same"))
43             model.add(Activation("relu"))
44             model.add(BatchNormalization(axis=chanDim))
45             model.add(MaxPooling2D(pool_size=(2, 2)))
46             model.add(Dropout(0.25))
```

The code above follows the *exact same pattern* as the above; however, now we are learning two sets of 64 filters (each of size 3×3) as opposed to 32 filters. Again, it is common to *increase* the number of filters as the spatial input size *decreases* deeper in the network.

Next comes our first (and only) set of `FC => RELU` layers:

```
48             # first (and only) set of FC => RELU layers
49             model.add(Flatten())
50             model.add(Dense(512))
51             model.add(Activation("relu"))
52             model.add(BatchNormalization())
53             model.add(Dropout(0.5))
```

Our `FC` layer has `512` nodes, which will be followed by a ReLU activation and BN. We'll also apply dropout here, increasing the probability to 50% – typically you'll see dropout with $p = 0.5$ applied in between `FC` layers.

Finally, we apply the softmax classifier and return the network architecture to the calling function:

```
55             # softmax classifier
56             model.add(Dense(classes))
57             model.add(Activation("softmax"))
58
59             # return the constructed network architecture
60             return model
```

Now that we've implemented the MiniVGGNet architecture, let's move on to applying it to CIFAR-10.

15.3 MiniVGGNet on CIFAR-10

We will follow a similar pattern training MiniVGGNet as we did for LeNet in Chapter 14, only this time with the CIFAR-10 dataset:

- Load the CIFAR-10 dataset from disk.
- Instantiate the MiniVGGNet architecture.
- Train MiniVGGNet using the training data.
- Evaluate network performance with the testing data.

To create a driver script to train MiniVGGNet, open a new file, name it `minivggnet_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 sklearn.metrics import classification_report
8  from pyimagesearch.nn.conv import MiniVGGNet
9  from keras.optimizers import SGD
10 from keras.datasets import cifar10
11 import matplotlib.pyplot as plt
12 import numpy as np
13 import argparse
```

Line 2 imports the `matplotlib` library which we'll later use to plot our accuracy and loss over time. We need to set the `matplotlib` backend to `Agg` to indicate to create a *non-interactive* that will simply be saved to disk. Depending on what your default `maplotlib` backend is *and* whether you are accessing your deep learning machine remotely (via SSH, for instance), X11 session may timeout. If that happens, `matplotlib` will error out when it tries to display your figure. Instead, we can simply set the background to `Agg` and write the plot to disk when we are done training our network.

Lines 9-13 import the rest of our required Python packages, all of which you've seen before – the exception being `MiniVGGNet` on **Line 11** which we implemented in the previous section.

Next, let's parse our command line arguments:

```
15 # construct the argument parse and parse the arguments
16 ap = argparse.ArgumentParser()
17 ap.add_argument("-o", "--output", required=True,
18     help="path to the output loss/accuracy plot")
19 args = vars(ap.parse_args())
```

This script will require only a single command line argument, `--output`, the path to our output training and loss plot.

We can now load the CIFAR-10 dataset (pre-split into training and testing data), scale the pixels into the range $[0, 1]$, and then one-hot encode the labels:

```
21 # load the training and testing data, then scale it into the
22 # range [0, 1]
23 print("[INFO] loading CIFAR-10 data...")
24 ((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()
trainY = lb.fit_transform(trainY)
testY = lb.transform(testY)

# initialize the label names for the CIFAR-10 dataset
labelNames = ["airplane", "automobile", "bird", "cat", "deer",
    "dog", "frog", "horse", "ship", "truck"]
```

Let's compile our model and start training MiniVGGNet:

```
# initialize the optimizer and model
print("[INFO] compiling model...")
opt = SGD(lr=0.01, decay=0.01 / 40, momentum=0.9, nesterov=True)
model = MiniVGGNet.build(width=32, height=32, depth=3, classes=10)
model.compile(loss="categorical_crossentropy", optimizer=opt,
    metrics=["accuracy"])

# train the network
print("[INFO] training network...")
H = model.fit(trainX, trainY, validation_data=(testX, testY),
    batch_size=64, epochs=40, verbose=1)
```

We'll use SGD as our optimizer with a learning rate of $\alpha = 0.1$ and momentum term of $\gamma = 0.9$. Setting `nestrov=True` indicates that we would like to apply Nestrov accelerated gradient to the SGD optimizer (Section 9.3).

An optimizer term we haven't seen yet is the `decay` parameter. This argument is used to slowly reduce the learning rate over time. As we'll discuss in more detail in the next chapter on *Learning Rate Schedulers*, decaying the learning rate is helpful in reducing overfitting and obtaining higher classification accuracy – the smaller the learning rate is, the smaller the weight updates will be. A common setting for `decay` is to divide the initial learning rate by the total number of epochs – in this case, we'll be training our network for a total of 40 epochs with an initial learning rate of 0.01, therefore `decay = 0.01 / 40`.

After training completes, we can evaluate the network and display a nicely formatted classification report:

```
# evaluate the network
print("[INFO] evaluating network...")
predictions = model.predict(testX, batch_size=64)
print(classification_report(testY.argmax(axis=1),
    predictions.argmax(axis=1), target_names=labelNames))
```

And with save our loss and accuracy plot to disk:

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

```
59 plt.plot(np.arange(0, 40), H.history["val_loss"], label="val_loss")
60 plt.plot(np.arange(0, 40), H.history["acc"], label="train_acc")
61 plt.plot(np.arange(0, 40), H.history["val_acc"], label="val_acc")
62 plt.title("Training Loss and Accuracy on CIFAR-10")
63 plt.xlabel("Epoch #")
64 plt.ylabel("Loss/Accuracy")
65 plt.legend()
66 plt.savefig(args["output"])
```

When evaluating MinIVGGNet I performed two experiments:

1. One *with* batch normalization.
2. One `without` batch normalization.

Let's go ahead and take a look at these results to compare how network performance increases when applying batch normalization.

15.3.1 With Batch Normalization

To train MiniVGGNet on the CIFAR-10 dataset, just execute the following command:

```
$ python minivggnet_cifar10.py --output output/cifar10_minivggnet_with_bn.png
[INFO] loading CIFAR-10 data...
[INFO] compiling model...
[INFO] training network...
Train on 50000 samples, validate on 10000 samples
Epoch 1/40
23s - loss: 1.6001 - acc: 0.4691 - val_loss: 1.3851 - val_acc: 0.5234
Epoch 2/40
23s - loss: 1.1237 - acc: 0.6079 - val_loss: 1.1925 - val_acc: 0.6139
Epoch 3/40
23s - loss: 0.9680 - acc: 0.6610 - val_loss: 0.8761 - val_acc: 0.6909
...
Epoch 40/40
23s - loss: 0.2557 - acc: 0.9087 - val_loss: 0.5634 - val_acc: 0.8236
[INFO] evaluating network...
             precision    recall  f1-score   support

   airplane       0.88      0.81      0.85      1000
 automobile       0.93      0.89      0.91      1000
       bird       0.83      0.68      0.75      1000
        cat       0.69      0.65      0.67      1000
       deer       0.74      0.85      0.79      1000
        dog       0.72      0.77      0.74      1000
       frog       0.85      0.89      0.87      1000
      horse       0.85      0.87      0.86      1000
       ship       0.89      0.91      0.90      1000
      truck       0.88      0.91      0.90      1000

avg / total       0.83      0.82      0.82     10000
```

On my GPU, epochs were quite fast at 23s. On my CPU, epochs were considerably longer, clocking in at 171s.

After training completed, we can see that MiniVGGNet is obtaining **83%** classification accuracy on the CIFAR-10 dataset *with* batch normalization – this result is substantially higher than the 60%

accuracy when applying ShallowNet in Chapter 12. We thus see how a deeper network architectures are able to learn richer, more discriminative features.

But what about the role of batch normalization? Is it actually helping us here? To find out, let's move on to the next section.

15.3.2 Without Batch Normalization

Go back to the `minivggnet.py` implementation and comment out *all* `BatchNormalization` layers, like so:

```
27          # first CONV => RELU => CONV => RELU => POOL layer set
28          model.add(Conv2D(32, (3, 3), padding="same",
29               input_shape=inputShape))
30          model.add(Activation("relu"))
31          #model.add(BatchNormalization(axis=chanDim))
32          model.add(Conv2D(32, (3, 3), padding="same"))
33          model.add(Activation("relu"))
34          #model.add(BatchNormalization(axis=chanDim))
35          model.add(MaxPooling2D(pool_size=(2, 2)))
36          model.add(Dropout(0.25))
```

Once you've commented out all `BatchNormalization` layers from your network, re-train MiniVGGNet on CIFAR-10:

```
$  python minivggnet_cifar10.py --output output/cifar10_minivggnet_without_bn.png
[INFO] loading CIFAR-10 data...
[INFO] compiling model...
[INFO] training network...
Train on 50000 samples, validate on 10000 samples
Epoch 1/40
13s - loss: 1.8055 - acc: 0.3426 - val_loss: 1.4872 - val_acc: 0.4573
Epoch 2/40
13s - loss: 1.4133 - acc: 0.4872 - val_loss: 1.3246 - val_acc: 0.5224
Epoch 3/40
13s - loss: 1.2162 - acc: 0.5628 - val_loss: 1.0807 - val_acc: 0.6139
...
Epoch 40/40
13s - loss: 0.2780 - acc: 0.8996 - val_loss: 0.6466 - val_acc: 0.7955
[INFO] evaluating network...
             precision    recall  f1-score   support

   airplane       0.83      0.80      0.82      1000
 automobile       0.90      0.89      0.90      1000
       bird       0.75      0.69      0.71      1000
        cat       0.64      0.57      0.61      1000
       deer       0.75      0.81      0.78      1000
        dog       0.69      0.72      0.70      1000
       frog       0.81      0.88      0.85      1000
      horse       0.85      0.83      0.84      1000
       ship       0.90      0.88      0.89      1000
      truck       0.84      0.89      0.86      1000

avg / total       0.79      0.80      0.79     10000
```

The first thing you'll notice is that your network trains *faster* without batch normalization (13s compared to 23s, a reduction by 43%). However, once the network finishes training, you'll notice a lower classification accuracy of **79%**.

When we plot MiniVGGNet *with* batch normalization (left) and *without* batch normalization (right) side-by-side in Figure 15.1, we can see the positive affect batch normalization has on the training process:

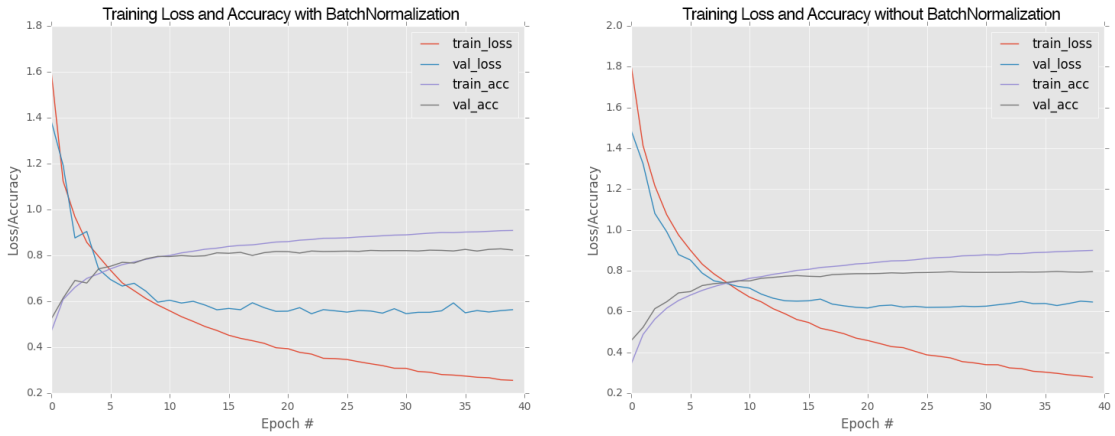

Figure 15.1: **Left:** MiniVGGNet trained on CIFAR-10 *with* batch normalization. **Right:** MiniVGGNet trained on CIFAR-10 *without* batch normalization. Applying batch normalization allows us to obtain higher classification accuracy and reduce the affects of overfitting.

Notice how the loss for MiniVGGNet without batch normalization *starts to increase* past epoch 30, indicating that the network is *overfitting* to the training data. We can also clearly see that validation accuracy has become quite saturated by epoch 25.

On the other hand, the MiniVGGNet implementation *with* batch normalization is more stable. While both loss and accuracy start to flatline past epoch 35, we aren't overfitting as badly – this is one of the many reasons why I suggest applying batch normalization to your own network architectures.

15.4 Summary

In this chapter we discussed the VGG family of Convolutional Neural Networks. A CNN can be considered VGG-net like if:

1. It makes use of *only* 3×3 filters, regardless of network depth.
2. There are *multiple* `CONV => RELU` layers applied *before* a single `POOL` operation, sometimes with more `CONV => RELU` layers stacked on top of each other as the network increases in depth.

We then implemented a VGG inspired network, suitably named `MiniVGGNet`. This network architecture consisted of two sets of `(CONV => RELU) * 2) => POOL` layers followed by an `FC => RELU => FC => SOFTMAX` layer set. We also applied batch normalization after every activation as well as dropout after every pool and fully-connected layer. To evaluate MiniVGGNet, we used the CIFAR-10 dataset.

Our previous best accuracy on CIFAR-10 was only 60% from the ShallowNet network (Chapter 12). However, using MiniVGGNet we were able to increase accuracy all the way to ***83%***.

Finally, we examined the role batch normalization plays in deep learning and CNNs *With* batch

normalization, MiniVGGNet reached 83% classification accuracy – but *without* batch normalization, accuracy decreased to 79% (and we also started to see signs of overfitting).

Thus, the takeaway here is that:

1. Batch normalization can lead to a faster, more stable convergence with higher accuracy.
2. However, the advantages will come at the expense of training time – batch normalization will require more "wall time" to train the network, even though the network will obtain higher accuracy in less epochs.

That said, the extra training time often outweighs the negatives, and I *highly encourage* you to apply batch normalization to your own network architectures.

16. Learning Rate Schedulers

In our last chapter, we trained the MiniVGGNet architecture on the CIFAR-10 dataset. To help alleviate the effects of overfitting, I introduced the concept of adding a *decay* to our learning rate when applying SGD to train the network.

In this chapter we'll discuss the concept of *learning rate schedules*, sometimes called learning rate annealing or adaptive learning rates. By adjusting our learning rate on an epoch-to-epoch basis, we can reduce loss, increase accuracy, and even in certain situations reduce the total amount of time it takes to train a network.

16.1 Dropping Our Learning Rate

The most simple and heavily learning rate schedulers are ones that progressively reduce learning rate over time. To consider why learning rate schedules are an interesting method to apply to help increase model accuracy, consider our standard weight update formula from Section 9.1.6:

```
W += -args["alpha"] * gradient
```

Recall that the learning rate α controls the "step" we make along the gradient. Larger values of α imply that we are taking bigger steps while smaller values of α will make tiny steps – if α is zero, the network cannot make any steps at all (since the gradient multiplied by zero is zero).

In our previous examples throughout this book, our learning rates were constant – we typically set $\alpha = \{0.1, 0.01\}$ and then trained the network for a fixed number of epochs without changing the learning rate. This method may work well in some situations, but it's often beneficial to *decrease* our learning rate over time.

When training our network, we are trying to find some location along our loss landscape where the network obtains reasonable accuracy. It doesn't have to be a global minima or even a local minima, but in practice, simply finding an area of the loss landscape with reasonably low loss is "good enough".

If we constantly keep a learning rate high, we could overshoot these areas of low loss as we'll be taking *too large* of steps to descend into these areas. Instead, what we can do is decrease our learning rate, thereby allowing our network to take smaller steps – this decreased rate enables our network to descend into areas of the loss landscape that are "more optimal" and would have otherwise been missed entirely by our larger learning rate.

We can, therefore, view the process of learning rate scheduling as:

1. Finding a set of reasonably "good" weights early in the training process with a higher learning rate.
2. Tuning these weights later in the process to find more optimal weights using a smaller learning rate.

There are two primary types of learning rate schedulers that you'll likely encounter:

1. Learning rate schedulers that decrease gradually based on the epoch number (like a linear, polynomial, or exponential function).
2. Learning rate schedulers that drop based on *specific* epoch (such as a piecewise function).

We'll review both types of learning rate schedulers in this chapter.

16.1.1 The Standard Decay Schedule in Keras

The Keras library ships with a time-based learning rate scheduler – it is controlled via the `decay` parameter of the optimizer classes (such as `SGD`).

Going back to our previous chapter, let's take a look at the code block where we initialize SGD and MiniVGGNet:

```
37 # initialize the optimizer and model
38 print("[INFO] compiling model...")
39 opt = SGD(lr=0.01, decay=0.01 / 40, momentum=0.9, nesterov=True)
40 model = MiniVGGNet.build(width=32, height=32, depth=3, classes=10)
41 model.compile(loss="categorical_crossentropy", optimizer=opt,
42     metrics=["accuracy"])
```

Here we initialize our `SGD` optimizer with a learning rate of $\alpha = 0.01$, a momentum $\gamma = 0.9$, and indicate that we are using Nesterov accelerated gradient. We then set our decay (γ) to be the learning rate divided by the total number of epochs we are training the network for (a common rule of thumb), resulting in $0.01/40 = 0.00025$.

Internally, Keras applies the following learning rate schedule to adjust the learning rate after *every epoch*:

$$\alpha_{e+1} = \alpha_e \times 1/(1+\gamma * e) \tag{16.1}$$

If we set the `decay` to zero (the default value in Keras optimizers unless we *explicitly* supply it), we'll notice there is no effect on the learning rate (here we arbitrarily set our current epoch e to be $e = 1$ to demonstrate this point):

$$\alpha_{e+1} = 0.01 \times 1/(1+0.0 \times 1) = 0.01 \tag{16.2}$$

But if we instead use `decay = 0.01 / 40`, you'll notice that the learning rate starts to decrease after every epoch (Table 16.1).

Using this time-based learning rate decay, our MiniVGGNet model obtained 83% classification accuracy, as shown in Chapter 15. I would encourage you to set `decay=0` in the SGD optimizer

Epoch	Learning Rate (α)
1	0.01
2	0.00990
3	0.00971
...	...
38	0.00685
39	0.00678
40	0.00672

Table 16.1: A table demonstrating how our learning rate decreases over time using 40 epochs, an initial learning rate of $\alpha = 0.01$ and a decay term of $0.04/40$.

and then rerun the experiment. You'll notice that the network also obtains $\approx 83\%$ classification accuracy; however, by investing the learning plots of the two models in Figure 16.1, you'll notice that overfitting is starting to occur as validation loss rises past epoch 25 (*left*).

This result is in contrast to when we set `decay=0.01 / 40` (*right*) and obtain a much nicer learning plot (and not to mention, higher accuracy). By using learning rate decay we can often not only improve our classification accuracy but also lessen the affects of overfitting, thereby increasing the ability of our model to generalize.

Figure 16.1: **Left:** Training MiniVGGNet on CIFAR-10 *without* learning rate decay. Notice how loss starts to increase past epoch 25, indicating that overfitting is happening. **Right:** Applying a decay factor of $0.01/40$. This reduces the learning rate over time, helping alleivate the affects of overfitting.

16.1.2 Step-based Decay

Another popular learning rate scheduler is step-based decay where we systematically drop the learning rate after *specific* epochs during training. The step decay learning rate schedulers can be thought of as a piecewise function, such as in Figure 16.2. Here the learning rate is constant for a number of epochs, then drops, and is constant once more, then drops again, etc.

When applying step decay to our learning rate, we have two options:

1. Define an equation that models the piecewise drop in learning rate we wish to achieve.

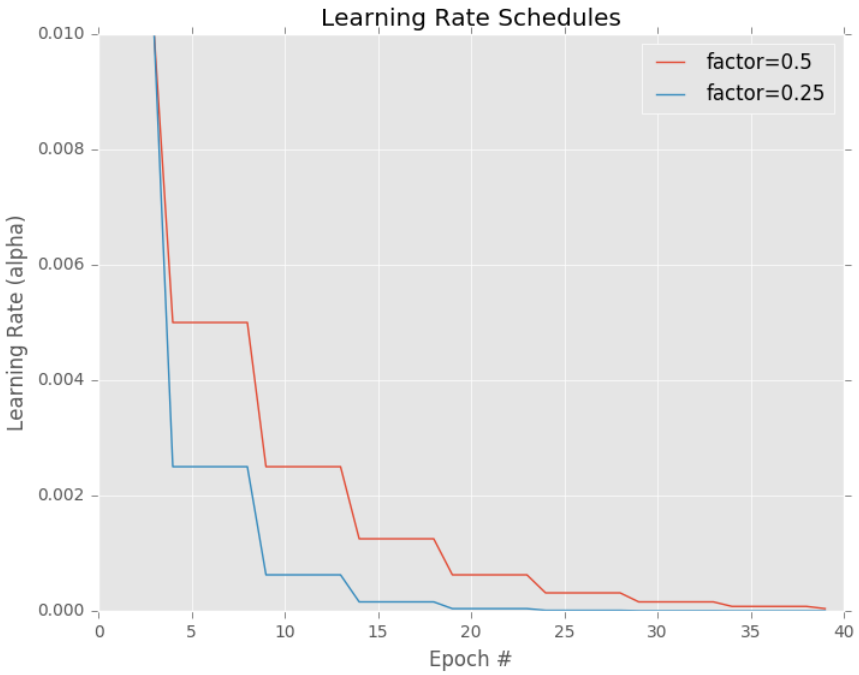

Figure 16.2: An example of two learning rate schedules that drop the learning rate in a piecewise fashion. Lowering the *factor* value increases the speed of the drop. In each case, the learning rate approaches zero at the final epoch.

2. Use what I call the `ctrl + c` method to training a deep learning network where we train for some number of epochs at a given learning rate, eventually notice validation performance has stalled, then `ctrl + c` to stop the script, adjust our learning rate, and continue training.

We'll primarily be focusing on the equation-based piecewise drop to learning rate scheduling in this chapter. The `ctrl + c` method is more advanced and is normally applied to larger datasets using deeper neural networks where the exact number of epochs required to obtain reasonable accuracy is unknown. I cover `ctrl + c` training *heavily* inside the *Practitioner Bundle* and *ImageNet Bundle* of this book.

When applying step decay, we often drop our learning rate by either (1) half or (2) an order of magnitude after every fixed number of epochs. For example, let's suppose our initial learning rate is $\alpha = 0.1$. After 10 epochs we drop the learning rate to $\alpha = 0.05$. After another 10 epochs of training (i.e., the 20th total epoch), α is dropped by 0.5 again, such that $\alpha = 0.025$, etc. In fact, this is the exact same learning rate schedule plotted in Figure 16.2 above (red line). The blue line displays a much more aggressive drop with a factor of 0.25.

16.1.3 Implementing Custom Learning Rate Schedules in Keras

Conveniently, the Keras library provides us with a `LearningRateScheduler` class that allows us to define a *custom learning rate function* and then have it *automatically applied* during the training process. This function should take the epoch number as an argument and then compute our desired learning rate based on a function that we define.

In this example, we'll be defining a piecewise function that will drop the learning rate by a

certain factor F after every D epochs. Our equation will thus look like this:

$$\alpha_{E+1} = \alpha_I \times F^{(1+E)/D} \tag{16.3}$$

Where α_I is our initial learning rate, F is the factor value controlling the rate in which the learning rate drops, D is the "drop every" epochs value, and E is the current epoch. The *larger* our factor F is, the *slower* the learning rate will decay. Conversely, the *smaller* the factor F is the *faster* the learning rate will decrease.

Written in Python code, this equation might be expressed as:

```
alpha = initAlpha * (factor ** np.floor((1 + epoch) / dropEvery))
```

Let's go ahead and implement this custom learning rate schedule and then apply it to MiniVGGNet on CIFAR-10. Open up a new file, name it `cifar10_lr_decay.py`, and let's start coding:

```
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 sklearn.metrics import classification_report
8  from pyimagesearch.nn.conv import MiniVGGNet
9  from keras.callbacks import LearningRateScheduler
10 from keras.optimizers import SGD
11 from keras.datasets import cifar10
12 import matplotlib.pyplot as plt
13 import numpy as np
14 import argparse
```

Lines 2-14 import our required Python packages as in the original `minivggnet_cifar10.py` script form Chapter 15. However, take notice of **Line 9** where we import our `LearningRateScheduler` from the Keras library – this class will enable us to define our own custom learning rate scheduler.

Speaking of a custom learning rate scheduler, let's define that now:

```
16 def step_decay(epoch):
17     # initialize the base initial learning rate, drop factor, and
18     # epochs to drop every
19     initAlpha = 0.01
20     factor = 0.25
21     dropEvery = 5
22
23     # compute learning rate for the current epoch
24     alpha = initAlpha * (factor ** np.floor((1 + epoch) / dropEvery))
25
26     # return the learning rate
27     return float(alpha)
```

Line 16 defines the `step_decay` function which accepts a single required parameter – the current `epoch`. We then define the initial learning rate (`0.01`), the drop factor (`0.25`), set `dropEvery = 5`, implying that we'll drop our learning rate by a factor of 0.25 every five epochs (**Lines 19-21**).

We compute the new learning rate for the current epoch on **Line 24** using the Equation 16.3 above. This new learning rate is returned to the calling function on **Line 27**, allowing Keras to internally update the optimizer's learning rate.

From here we can continue on with our script:

```
29 # construct the argument parse and parse the arguments
30 ap = argparse.ArgumentParser()
31 ap.add_argument("-o", "--output", required=True,
32     help="path to the output loss/accuracy plot")
33 args = vars(ap.parse_args())
34
35 # load the training and testing data, then scale it into the
36 # range [0, 1]
37 print("[INFO] loading CIFAR-10 data...")
38 ((trainX, trainY), (testX, testY)) = cifar10.load_data()
39 trainX = trainX.astype("float") / 255.0
40 testX = testX.astype("float") / 255.0
41
42 # convert the labels from integers to vectors
43 lb = LabelBinarizer()
44 trainY = lb.fit_transform(trainY)
45 testY = lb.transform(testY)
46
47 # initialize the label names for the CIFAR-10 dataset
48 labelNames = ["airplane", "automobile", "bird", "cat", "deer",
49     "dog", "frog", "horse", "ship", "truck"]
```

Lines 30-33 parse our command line arguments. We only need a single argument, `--output`, the path to our output loss/accuracy plot. We then load the CIFAR-10 dataset from disk and scale the pixel intensities to the range $[0, 1]$ on **Lines 37-40**. **Lines 43-45** handle one-hot encoding the class labels.

Next, let's train our network:

```
51 # define the set of callbacks to be passed to the model during
52 # training
53 callbacks = [LearningRateScheduler(step_decay)]
54
55 # initialize the optimizer and model
56 opt = SGD(lr=0.01, momentum=0.9, nesterov=True)
57 model = MiniVGGNet.build(width=32, height=32, depth=3, classes=10)
58 model.compile(loss="categorical_crossentropy", optimizer=opt,
59     metrics=["accuracy"])
60
61 # train the network
62 H = model.fit(trainX, trainY, validation_data=(testX, testY),
63     batch_size=64, epochs=40, callbacks=callbacks, verbose=1)
```

Line 53 is important as it initializes our list of `callbacks`. Depending on how the callback is defined, Keras will call this function at the start or end of every epoch, mini-batch update, etc. The `LearningRateScheduler` will call `step_decay` at the end of every epoch, allowing us to update the learning prior to the *next* epoch starting.

Line 56 initializes the `SGD` optimizer with a momentum of `0.9` and Nestrov accelerated gradient. The `lr` parameter will be ignored here since we'll be using the `LearningRateScheduler` callback so we could *technically* leave this parameter out entirely; however, I like to include it and have it match `initAlpha` as a matter of clarity.

Line 57 initializes MiniVGGNet which we then train for 40 epochs on **Lines 62 and 63**.

Once the network is trained we can evaluate it:

```
65 # evaluate the network
66 print("[INFO] evaluating network...")
67 predictions = model.predict(testX, batch_size=64)
68 print(classification_report(testY.argmax(axis=1),
69     predictions.argmax(axis=1), target_names=labelNames))
```

As well as plot the loss and accuracy:

```
71 # plot the training loss and accuracy
72 plt.style.use("ggplot")
73 plt.figure()
74 plt.plot(np.arange(0, 40), H.history["loss"], label="train_loss")
75 plt.plot(np.arange(0, 40), H.history["val_loss"], label="val_loss")
76 plt.plot(np.arange(0, 40), H.history["acc"], label="train_acc")
77 plt.plot(np.arange(0, 40), H.history["val_acc"], label="val_acc")
78 plt.title("Training Loss and Accuracy on CIFAR-10")
79 plt.xlabel("Epoch #")
80 plt.ylabel("Loss/Accuracy")
81 plt.legend()
82 plt.savefig(args["output"])
```

To evaluate the effect the drop factor has on learning rate scheduling and overall network classification accuracy, we'll be evaluating two drop factors: 0.25 and 0.5, respectively. The drop factor of 0.25 will decrease significantly faster than the 0.5 rate, as we know from Figure 16.2 above.

Again, take notice how much faster the 0.25 drop factor lowers our learning rate. We'll go ahead and evaluate the faster learning rate drop of 0.25 (**Line 20**) – to execute our script, just issue the following command:

```
$ python cifar10_lr_decay.py --output output/lr_decay_f0.25_plot.png
[INFO] loading CIFAR-10 data...
Train on 50000 samples, validate on 10000 samples
Epoch 1/40
34s - loss: 1.6380 - acc: 0.4550 - val_loss: 1.1413 - val_acc: 0.5993
Epoch 2/40
34s - loss: 1.1847 - acc: 0.5925 - val_loss: 1.0986 - val_acc: 0.6057
...
Epoch 40/40
34s - loss: 0.5423 - acc: 0.8081 - val_loss: 0.5899 - val_acc: 0.7885
[INFO] evaluating network...
             precision    recall  f1-score   support

   airplane       0.81      0.81      0.81      1000
 automobile       0.91      0.89      0.90      1000
```

```
        bird       0.71      0.65      0.68      1000
         cat       0.63      0.60      0.62      1000
        deer       0.72      0.79      0.75      1000
         dog       0.70      0.67      0.68      1000
        frog       0.80      0.88      0.84      1000
       horse       0.86      0.83      0.84      1000
        ship       0.87      0.90      0.88      1000
       truck       0.87      0.87      0.87      1000

   avg / total       0.79      0.79      0.79     10000
```

Here we see that our network obtains only 79% classification accuracy. The learning rate is dropping quite aggressively – after epoch fifteen α is only 0.00125, meaning that our network is taking very small steps along the loss landscape. This behavior can be seen in the Figure 16.3 (*left*) where validation loss and accuracy have essentially stagnated after epoch fifteen.

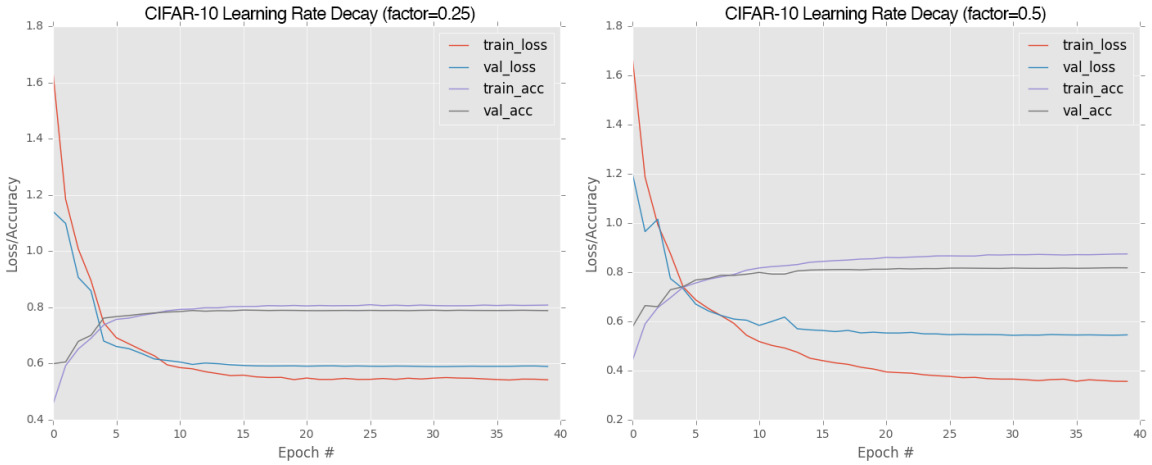

Figure 16.3: **Left:** Plotting the accuracy/loss of our network using a faster learning rate drop with a factor of 0.25. Notice how loss/accuracy stagnate past epoch 15 as the learning rate is too small. **Right:** Accuracy/loss of our network with a slower learning rate drop ($factor = 0.5$). This time our network is able to continue to learn past epoch 30 until stagnation occurs.

If we instead change the drop factor to 0.5 by setting `factor = 0.5` inside of `step_decay`:

```
16 def step_decay(epoch):
17     # initialize the base initial learning rate, drop factor, and
18     # epochs to drop every
19     initAlpha = 0.01
20     factor = 0.5
21     dropEvery = 5
```

And then re-run the experiment, we'll obtain higher classification accuracy:

```
$ python cifar10_lr_decay.py --output output/lr_decay_f0.5_plot.png
[INFO] loading CIFAR-10 data...
Train on 50000 samples, validate on 10000 samples
```

```
Epoch 1/40
35s - loss: 1.6733 - acc: 0.4402 - val_loss: 1.2024 - val_acc: 0.5771
Epoch 2/40
34s - loss: 1.1868 - acc: 0.5898 - val_loss: 0.9651 - val_acc: 0.6643
...
Epoch 40/40
33s - loss: 0.3562 - acc: 0.8742 - val_loss: 0.5452 - val_acc: 0.8177
[INFO] evaluating network...
             precision    recall  f1-score   support

   airplane       0.85      0.82      0.84      1000
 automobile       0.91      0.91      0.91      1000
       bird       0.75      0.70      0.73      1000
        cat       0.68      0.65      0.66      1000
       deer       0.75      0.82      0.78      1000
        dog       0.74      0.74      0.74      1000
       frog       0.83      0.89      0.86      1000
      horse       0.88      0.86      0.87      1000
       ship       0.89      0.91      0.90      1000
      truck       0.89      0.88      0.88      1000

avg / total       0.82      0.82      0.82     10000
```

This time, with the slower drop in learning rate we obtain 82% accuracy. Looking at the plot in Figure 16.3 (*right*) we can see that our network continues to learn past epoch 25-30 until loss stagnates on the validation data Past epoch 30 the learning rate is very small at `2.44e-06` and is unable to make any significant changes to the weights to influence the loss/accuracy on the validation data.

16.2 Summary

The purpose of this chapter was to review the concept of *learning rate schedulers* and how they can be used to increase classification accuracy. We discussed the two primary types of learning rate schedulers:

1. Time-based schedulers that gradually decrease based on epoch number.
2. Drop-based schedulers that drop based on a *specific* epoch, similar to the behavior of a piecewise function.

Exactly *which* learning rate scheduler you should use (if you should use a scheduler at all) is part of the experimentation process. Typically your first experiment *would not* use any type of decay or learning rate scheduling so you can obtain a *baseline* accuracy and loss/accuracy curve.

From there you might introduce the standard time-based schedule provided by Keras (with the rule of thumb of `decay = alpha_init / epochs`) and run a second experiment to evaluate the results. The next few experiments might involve swapping out a time-bases schedule for a drop-based one using various drop factors.

Depending on how challenging your classification dataset is along with the depth of your network, you might opt for the `ctrl + c` method to training as detailed in the *Practitioner Bundle* and *ImageNet Bundle* which is the approach taken by most deep learning practitioners when training networks on the ImageNet dataset.

Overall, be prepared to spend a *significant amount of time* training your networks and evaluating different sets of parameters and learning routines. Even simple datasets and projects can take 10's to 100's of experiments to obtain a high accuracy model.

At this point in your study of deep learning you should understand that training deep neural networks is part science, part art. My goal in this book is to provide you with the science behind training a network along with the common rules of thumb that I use so you can learn the "art" behind it – but keep in mind that *nothing* beats *actually running the experiments yourself*.

The more practice you have at training neural networks, logging the results of what did work and what didn't, the better you'll become at it. There is *no shortcut* when it comes to mastering this art – you need to put in the hours and become comfortable with the SGD optimizer (and others) along with their parameters.

17. Spotting Underfitting and Overfitting

We briefly touched on underfitting and overfitting in Chapter 9. We are now going to take a deeper dive and discuss both underfitting and overfitting in the context of deep learning. To help us understand the concept of both underfitting and overfitting, I'll provide a number of graphs and figures so you can match your own training loss/accuracy curves to them – this practice will be especially useful if this book is your first exposure to machine learning/deep learning and you haven't had to spot underfitting/overfitting before.

From there we'll discuss how we can create a *(near) real-time training monitor* for Keras that you can use to babysit the training process of your network. Up until now, we've had to wait until *after* our network had completed training before we could plot the training loss and accuracy.

Waiting until the *end* of the training process to visualize loss and accuracy can be computationally wasteful, *especially* if our experiments take a long time to run and we have *no way* to visualize loss/accuracy during the training process itself (other than looking at the raw terminal output) – we could spend hours or even days training a network when without realizing that the process should have been stopped after the first few epochs.

Instead, it would be much more beneficial if we could plot the training and loss after *every epoch* and visualize the results. From there we could make better, more informed decisions regarding whether we should terminate the experiment early or keep training.

17.1 What Are Underfitting and Overfitting?

When training your own neural networks, you need to be *highly concerned* with both underfitting and overfitting. ***Underfitting*** occurs when your model cannot obtain sufficiently low loss on the *training set*. In this case, your model fails to learn the underlying patterns in your training data. On the other end of the spectrum, we have **overfitting** where your network models the training data *too well* and fails to generalize to your validation data.

Therefore, our goal when training a machine learning model is to:

1. Reduce the training loss as much as possible.
2. While ensuring the gap between the training and testing loss is reasonably small.

Controlling whether a model is likely to underfit or overfit can be accomplished by adjusting the *capacity* of the neural network. We can *increase* capacity by adding more layers and neurons to our network. Similarly, we can *decrease* capacity by removing layers and neurons and applying regularization techniques (weight decay, dropout, data augmentation, early stopping, etc.).

The following Figure 17.1, (inspired by the excellent example of Figure 5.3 of Goodfellow et al., page 112 [10]), helps visualize the relationship between underfitting and overfitting in conjunction with model capacity:

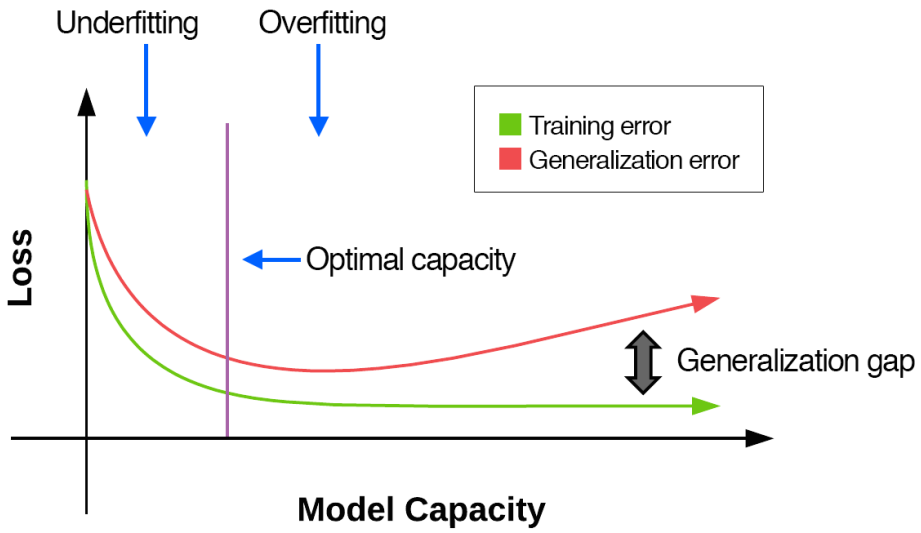

Figure 17.1: Relationship between model capacity and loss. The vertical purple line separates optimal capacity from underfitting (left) and overfitting (right). When we are underfitting the generalization gap is maintained. Optimal capacity occurs when the loss for both training and generalization levels out. When generalization loss increases we are overfitting. *Note: Figure inspired by Goodfellow et al., page 112 [10].*

On the x-axis, we have plotted the capacity of the network, and on the y-axis, we have the loss, where lower loss is more desirable. Typically when a network starts training we are in the "underfitting zone" (Figure 17.1, *left*). At this point, we are simply trying to learn some initial patterns in the underlying data and move the weights away from their random initializations to values that enable us to actually "learn" from the data itself. Ideally, both the training loss and validation loss will drop together during this period – that drop demonstrates that our network is actually learning.

However, as our model capacity increases (due to deeper networks, more neurons, no regularization, etc.) we'll reach the "optimal capacity" of the network. At this point, our training and validation loss/accuracy start to diverge from each other, and a noticeable gap starts to form. Our goal is to *limit this gap*, thus preserving the generalizability of our model.

If we fail to limit this gap, we enter the "overfitting zone" (Figure 17.1, *right*). At this point, our training loss will either stagnate/continue to drop while our validation loss stagnates and eventually *increases*. An increase in validation loss over a series of consecutive epochs is a heavy indicator of overfitting.

So, how do we combat overfitting? In general, there are two techniques:

1. Reduce the complexity of the model, opting for a more shallow network with less layers and

neurons.

2. Apply regularization methods.

Using smaller neural networks may work for smaller datasets, but in general, this is *not* the preferred solution. Instead, we should apply regularization techniques such as weight decay, dropout, data augmentation, etc. In practice, it's nearly always better to use regularization techniques to control overfitting rather than your network size [129] *unless* you have very good reason to believe that your network architecture is simply *far too large* for the problem.

17.1.1 Effects of Learning Rates

In our previous section we looked at an example of overfitting – but what role does our learning rate play in overfitting? Is there actually a way to *spot* if our model is overfitting given a set of hyperparameters simply by examining the loss curve? You bet there is. Just take a look at Figure 17.2 for an example (heavily inspired by Karpathy et al. [93]).

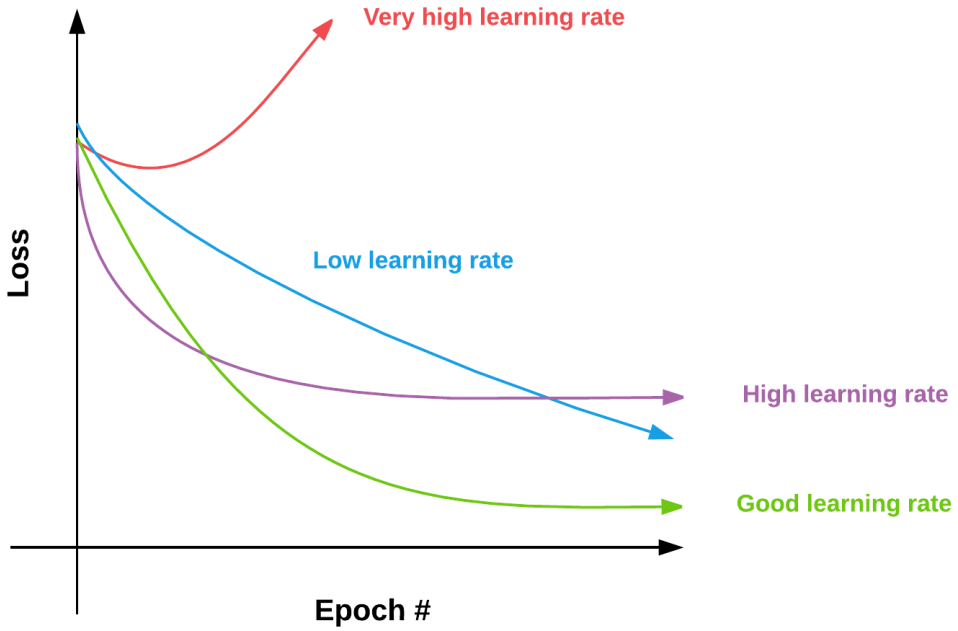

Figure 17.2: A plot visualizing the affect varying learning rates will have on loss (heavily inspired by Karpathy et al. [93]). Very high learning rates (**red**) will drop loss initially but dramatically increase soon after. Low learning rates (**blue**) will be approximately linear in their loss over time while high learning rates (**purple**) will drop quickly, but level-off. Finally, good learning rates (**green**) decrease will decrease at a rate faster than linear, but at a lower exponential, allowing us to navigate the loss landscape.

On the x-axis, we have plotted the *epochs* of a neural network along with the corresponding *loss* on the y-axis. Ideally, our training loss and validation loss should look like the green curve, where loss drops quickly but not *so quickly* that we are unable to navigate our loss landscape and settle into an area of low loss.

Furthermore, when we plot both the training and validation loss at the *same time* we can get an even more detailed understanding of training progress. Preferably, our training and validation loss would nearly mimic each other, with only a small gap between training loss and validation loss, indicating little overfitting.

However, in many real-world applications, identical, mimicking behavior is simply not practical or even desirable as it may imply that it will take a long time to train our model. Therefore, we simply need to "mind the gap" between the training and validation loss. As long as the gap doesn't increase dramatically, we know there is an acceptable level of overfitting. However, if we fail to maintain this gap and training and validation loss diverge heavily, then we know we are at risk of overfitting. Once the validation loss starts to increase, we know we are *strongly overfitting*.

17.1.2 Pay Attention to Your Training Curves

When training your own neural networks, *pay attention to your loss and accuracy curves* for both the training data and validation data. During the first few epochs, it may seem that a neural network is tracking well, perhaps underfitting slightly – but this pattern can change quickly, and you might start to see a divergence in training and validation loss. When this happens, assess your model:

- Are you applying any regularization techniques?
- Is your learning rate too high?
- Is your network *too* deep?

Again, training deep learning networks is part science, part art. The best way to learn how to read these curves is to train as many networks as you can and inspect their plots. Over time you *will* develop a sense of what works and what doesn't – but don't expect to get it "right" on the first try. Even the most seasoned deep learning practitioner will run 10's to 100's of experiments, examining the loss/accuracy curves along the way, noting what works and what doesn't, and eventually zeroing in on a solution that works.

Finally, you should also accept the fact that overfitting for certain datasets is an *inevitability*. For example, it is *very easy* to overfit a model to the CIFAR-10 dataset. If your training loss and validation loss start to diverge, don't panic – simply try to control the gap as much as possible.

Also realize that as you lower your learning rate in later epochs (such as when using a learning rate scheduler), it will become easier to overfit as well. This point will become more clear in the more advanced chapters in both the *Practitioner Bundle* and *ImageNet Bundle*.

17.1.3 What if Validation Loss Is Lower than Training Loss?

Another strange phenomenon you might encounter is when your validation loss is actually *lower* than your training loss. How is this possible? How can a network possibly be performing better on the *validation data* when the patterns it is trying to learn is from the *training data*? Shouldn't the training performance *always* be better than the validation or testing loss?

Not always. In fact, there are multiple reasons for this behavior. Perhaps the simplest explanation is:

1. Your training data is seeing all the "hard" examples to classify.
2. While your validation data consists of the "easy" data points.

However, unless you *purposely* sampled your data in this way, it's unlikely that a random training and testing split would neatly segment these types of data points.

A second reason would be *data augmentation*. We cover data augmentation in detail inside the *Practitioner Bundle*, but the gist is that during the training process we randomly alter the training images by applying random transformations to them such as translation, rotation, resizing, and shearing. Because of these alterations, the network is constantly seeing augmented examples of the training data, which is a form of regularization, enabling the network to generalize better to the validation data while perhaps performing worse on the training set (see Chapter 9.4 for more details on regularization).

A third reason could be you're not training "hard enough". You might want to consider increasing your learning rate and tweaking your regularization strength.

17.2 Monitoring the Training Process

In the first part of this section, we'll create a `TrainingMonitor` callback that will be called at the end of every epoch when training a network with Keras. This monitor will serialize the loss and accuracy for both the training and validation set to disk, followed by constructing a plot of the data.

Applying this callback during training will enable us to babysit the training process and spot overfitting early, allowing us to abort the experiment and continue trying to tune our parameters.

17.2.1 Creating a Training Monitor

Our `TrainingMonitor` class will live in the `pyimagesearch.callbacks` sub-module:

```
|--- pyimagesearch
|    |--- __init__.py
|    |--- callbacks
|    |    |--- __init__.py
|    |    |--- trainingmonitor.py
|    |--- datasets
|    |--- nn
|    |--- preprocessing
|    |--- utils
```

Create the `trainingmonitor.py` file and let's get started:

```
1  # import the necessary packages
2  from keras.callbacks import BaseLogger
3  import matplotlib.pyplot as plt
4  import numpy as np
5  import json
6  import os
7
8  class TrainingMonitor(BaseLogger):
9      def __init__(self, figPath, jsonPath=None, startAt=0):
10         # store the output path for the figure, the path to the JSON
11         # serialized file, and the starting epoch
12         super(TrainingMonitor, self).__init__()
13         self.figPath = figPath
14         self.jsonPath = jsonPath
15         self.startAt = startAt
```

Lines 1-6 import our required Python packages. In order to create a class that logs our loss and accuracy to disk, we'll need to extend Keras' `BaseLogger` class (**Line 2**).

The constructor to the `TrainingMonitor` class is defined on **Line 9**. The constructor requires one argument, followed by two optional ones:

- `figPath`: The path to the output plot that we can use to visualize loss and accuracy over time.
- `jsonPath`: An optional path used to serialize the loss and accuracy values as a JSON file. This path is useful if you want to use the training history to create custom plots of your own.
- `startAt`: This is the starting epoch that training is resumed at when using `ctrl + c` training. We cover `ctrl + c` training in the *Practitioner Bundle* so we can ignore this variable for now.

Next, let's define the `on_train_begin` callback, which as the name suggests, is called *once* when the training process starts:

```
17      def on_train_begin(self, logs={}):
18          # initialize the history dictionary
19          self.H = {}
20
21          # if the JSON history path exists, load the training history
22          if self.jsonPath is not None:
23              if os.path.exists(self.jsonPath):
24                  self.H = json.loads(open(self.jsonPath).read())
25
26                  # check to see if a starting epoch was supplied
27                  if self.startAt > 0:
28                      # loop over the entries in the history log and
29                      # trim any entries that are past the starting
30                      # epoch
31                      for k in self.H.keys():
32                          self.H[k] = self.H[k][:self.startAt]
```

On **Line 19** we define `H`, used to represent the "history" of losses. We'll see how this dictionary is updated in the `on_epoch_end` function in the next code block.

Line 22 makes a check to see if a JSON path was supplied. If so, we then check to see if this JSON file exists. Provided that the JSON file does exist, we load its contents and update the history dictionary `H` *up until* the starting epoch (since that is where we will resume training from).

We can now move on to the most important function, `on_epoch_end` which is called when a training epoch completes:

```
34      def on_epoch_end(self, epoch, logs={}):
35          # loop over the logs and update the loss, accuracy, etc.
36          # for the entire training process
37          for (k, v) in logs.items():
38              l = self.H.get(k, [])
39              l.append(v)
40              self.H[k] = l
```

The `on_epoch_end` method is *automatically supplied* to parameters from Keras. The first is an integer representing the `epoch` number. The second is a dictionary, `logs`, which contains the training and validation loss + accuracy for the current epoch. We loop over each of the items in `logs` and then update our history dictionary (**Lines 37-40**).

After this code executes, the dictionary `H` now has four keys:

1. `train_loss`
2. `train_acc`
3. `val_loss`
4. `val_acc`

We maintain a *list* of values for each of these keys. Each list is updated at the end of *every epoch*, thus enabling us to plot an updated loss and accuracy curve as soon as the epoch completes.

In the case that a `jsonPath` was provided, we serialize the history `H` to disk:

```
42          # check to see if the training history should be serialized
43          # to file
44          if self.jsonPath is not None:
45              f = open(self.jsonPath, "w")
```

```
46                 f.write(json.dumps(self.H))
47                 f.close()
```

Finally, we can construct the actual plot as well:

```
49             # ensure at least two epochs have passed before plotting
50             # (epoch starts at zero)
51             if len(self.H["loss"]) > 1:
52                 # plot the training loss and accuracy
53                 N = np.arange(0, len(self.H["loss"]))
54                 plt.style.use("ggplot")
55                 plt.figure()
56                 plt.plot(N, self.H["loss"], label="train_loss")
57                 plt.plot(N, self.H["val_loss"], label="val_loss")
58                 plt.plot(N, self.H["acc"], label="train_acc")
59                 plt.plot(N, self.H["val_acc"], label="val_acc")
60                 plt.title("Training Loss and Accuracy [Epoch {}]".format(
61                     len(self.H["loss"])))
62                 plt.xlabel("Epoch #")
63                 plt.ylabel("Loss/Accuracy")
64                 plt.legend()
65
66                 # save the figure
67                 plt.savefig(self.figPath)
68                 plt.close()
```

Now that our `TrainingMonitor` is defined, let's move on to actually *monitoring* and *babysitting* the training process.

17.2.2 Babysitting Training

To monitor the training process, we'll need to create a driver script that trains a network using the `TrainingMonitor` callback. To begin, open up a new file, name it `cifar10_monitor.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 pyimagesearch.callbacks import TrainingMonitor
7  from sklearn.preprocessing import LabelBinarizer
8  from pyimagesearch.nn.conv import MiniVGGNet
9  from keras.optimizers import SGD
10 from keras.datasets import cifar10
11 import argparse
12 import os
```

Lines 1-12 import our required Python packages. Note how we are importing our newly defined `TrainingMonitor` class to enable us to babysit the training of our network.

Next, we can parse our command line arguments:

```
14 # construct the argument parse and parse the arguments
15 ap = argparse.ArgumentParser()
16 ap.add_argument("-o", "--output", required=True,
17     help="path to the output directory")
18 args = vars(ap.parse_args())
19
20 # show information on the process ID
21 print("[INFO process ID: {}".format(os.getpid()))
```

The only command line argument we need is `--output`, the path to the output directory to store our `matplotlib` generated figure and serialized JSON training history.

A neat trick I like to do is use the process ID assigned by the operating system to name my plots and JSON files. If I notice that training is going poorly, I can simply open up my task manager and kill the process ID associated with my script. This ability is especially useful if you are running *multiple experiments* at the same time. **Line 21** simply displays the process ID to our screen.

From there, we perform our standard pipeline of loading the CIFAR-10 dataset and preparing the data + labels for training:

```
23 # load the training and testing data, then scale it into the
24 # range [0, 1]
25 print("[INFO] loading CIFAR-10 data...")
26 ((trainX, trainY), (testX, testY)) = cifar10.load_data()
27 trainX = trainX.astype("float") / 255.0
28 testX = testX.astype("float") / 255.0
29
30 # convert the labels from integers to vectors
31 lb = LabelBinarizer()
32 trainY = lb.fit_transform(trainY)
33 testY = lb.transform(testY)
34
35 # initialize the label names for the CIFAR-10 dataset
36 labelNames = ["airplane", "automobile", "bird", "cat", "deer",
37     "dog", "frog", "horse", "ship", "truck"]
```

We are now ready to initialize the SGD optimizer along with the MiniVGGNet architecture:

```
39 # initialize the SGD optimizer, but without any learning rate decay
40 print("[INFO] compiling model...")
41 opt = SGD(lr=0.01, momentum=0.9, nesterov=True)
42 model = MiniVGGNet.build(width=32, height=32, depth=3, classes=10)
43 model.compile(loss="categorical_crossentropy", optimizer=opt,
44     metrics=["accuracy"])
```

Notice how I am *not* including a learning rate decay of any sort. This omission is *on purpose* so I can demonstrate how to monitor your training process and spot overfitting *as it's happening*.

Let's construct our `TrainingMonitor` callback and train the network:

```
46 # construct the set of callbacks
47 figPath = os.path.sep.join([args["output"], "{}.png".format(
48     os.getpid())])
```

```
49 jsonPath = os.path.sep.join([args["output"], "{}.json".format(
50      os.getpid())])
51 callbacks = [TrainingMonitor(figPath, jsonPath=jsonPath)]
52
53 # train the network
54 print("[INFO] training network...")
55 model.fit(trainX, trainY, validation_data=(testX, testY),
56      batch_size=64, epochs=100, callbacks=callbacks, verbose=1)
```

Lines 47-50 initialize the paths to our output plot and JSON serialized file, respectively. Notice how *each* of these file paths includes the process ID, allowing us to easily associate an experiment with a process ID – in the case that an experiment goes poorly, we can kill the script off using our task manager. Given the figure and JSON paths, **Line 51** builds our `callbacks` list, consisting of a single entry, the `TrainingMonitor` itself.

Finally, **Lines 55 and 56** train our network for a total of 100 epochs. I've purposely set the epochs very high to encourage our network to overfit.

To execute the script (and learn how to spot overfitting), just execute the following command:

```
$ python cifar10_monitor.py --output output
```

After the first few epochs you'll notice two files in your `--output` directory:

```
$ ls output/
7857.json  7857.png
```

These are your serialized training history and learning plots, respectively. Each file is named after the process ID that created them. The benefit of using the `TrainingMonitor` is that I can now **babysit the learning process** and monitor the training after every epoch completes.

For example, Figure 17.3 (*top-left*) displays our loss and accuracy plot after **epoch 5**. Right now we are still in the "underfitting zone". Our network is clearly learning from the training data as we can see loss decreasing and accuracy increasing; however, we have not reached any plateaus.

After **epoch 10** we can notice signs of overfitting, but nothing that is overly alarming (Figure 17.3, *top-middle*). The training loss is starting to diverge from the validation loss, but some divergence is entirely normal, and even a good indication that our network is continuing to learn underlying patterns from the training data.

However, by **epoch 25** we have reached the "overfitting zone" (Figure 17.3, *top-right*). Training loss is continuing to decline while validation loss has stagnated. This is a clear first sign of overfitting and bad things to come.

By **epoch 50** we are clearly in trouble (Figure 17.3, *bottom-left*). Validation loss is starting to *increase*, the **tell-tale sign of overfitting**. At this point, you should have definitely stopped the experiment to reassess your parameters.

If we were to let the network train until **epoch 100**, the overfitting would only get worse (Figure 17.3, *bottom-right*). The gap between training loss and validation loss is *gigantic*, all while validation loss continues to increase. While the validation accuracy of this network is above 80%, the ability of this model to generalize would be quite poor. Based on these plots we can clearly see when and where overfitting starts to occur. When running your own experiments, make sure you use the `TrainingMonitor` to aid you in babysitting the training process.

Finally, when you start to *think* there are signs of overfitting, don't become too trigger happy to kill off the experiment. Let the network train for another 10-15 epochs to *ensure* your hunch is

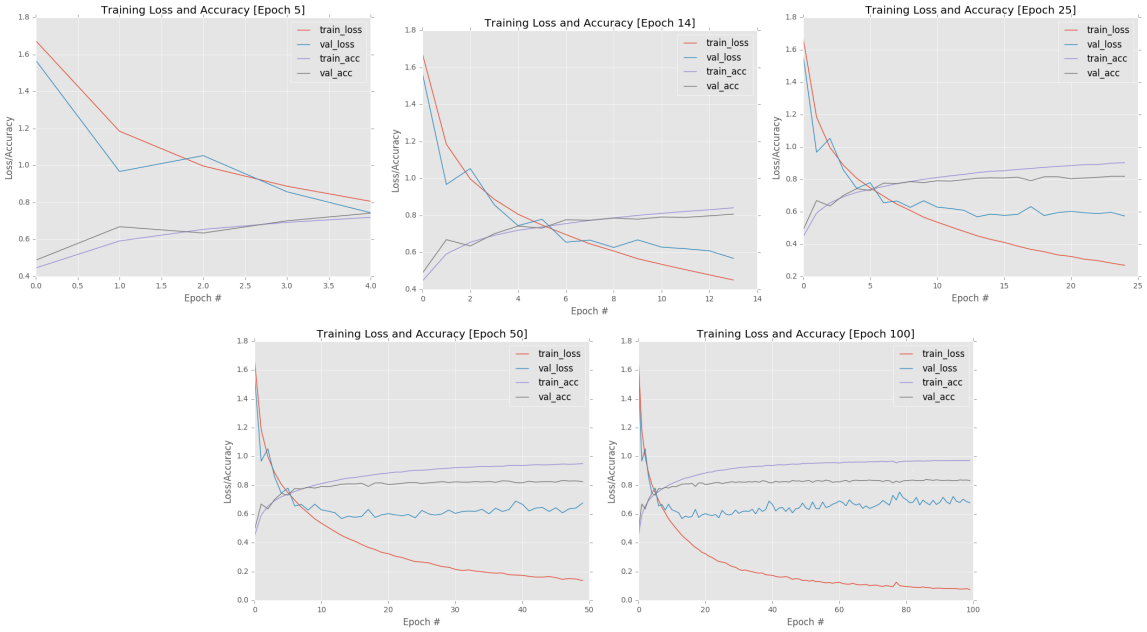

Figure 17.3: Examples of monitoring the training process by examining the training/validation loss curves. At epoch 5 we are still in the underfitting zone. At epoch 14 we are starting to overfit, but nothing to be overly concerned about. By epoch 25 we are certainly inside the overfitting zone. Epochs 50 and 100 demonstrate heavy overfitting as the validation loss rises while training loss continues to fall.

correct and that overfitting is occurring – we often need the *context* of these epochs to help us make this final decision.

Too often I see deep learning practitioners new to machine learning too trigger happy and kill experiments too early. Wait until you see clear signs of overfitting, then kill the process. As you hone your deep learning skills you'll develop a sixth sense to guide you when training your networks, but until then, trust the context of the extra epochs to enable you to make a better, more informed decision.

17.3 Summary

In this chapter we reviewed underfitting and overfitting. Underfitting occurs when your model is unable to obtain sufficiently low loss on the *training* set. Meanwhile, overfitting occurs when the gap between your training loss and validation loss is *too large*, indicating that the network is modeling the underlying patterns in the training data too strong.

Underfitting is relatively easy to combat: simply add more layers/neurons to your network. Overfitting is an entirely different beast though. When overfitting occurs you should consider:

1. Reducing the capacity of your network by removing layers/neurons (not recommended unless for small dataset).
2. Applying stronger regularization techniques.

In nearly all situations you should first attempt applying stronger regularization than reducing the size of your network – the exception being if you're attempting to train a massively deep network on a tiny dataset.

After understanding the relationship between model capacity and both underfitting and overfitting, we learned how to monitor our training process and spot overfitting *as it's happening* – this

process allows us to stop networks from training early instead of wasting time letting the network overfit. Finally, we wrapped up the chapter by looking at some tell-tale examples of overfitting.

18. Checkpointing Models

In Chapter 13 we discussed how to save and serialize your models to disk after training is complete. And in the last chapter, we learned how to spot underfitting and overfitting *as they are happening*, enabling you to kill off experiments that are not performing well while keeping the models that show promise while training.

However, you might be wondering if it's possible to *combine* both of these strategies. Can we serialize models whenever our loss/accuracy improves? Or is it possible to serialize only the *best* model (i.e., the one with the lowest loss or highest accuracy) during the training process? You bet. And luckily, we don't have to build a custom callback either – this functionality is baked right into Keras.

18.1 Checkpointing Neural Network Model Improvements

A good application of checkpointing is to serialize your network to disk each time there is an improvement during training. We define an "improvement" to be either a *decrease* in loss or an *increase* in accuracy – we'll set this parameter inside the actual Keras callback.

In this example, we'll be training the MiniVGGNet architecture on the CIFAR-10 dataset and then serializing our network weights to disk each time model performance improves. To get started, open up a new file, name it `cifar10_checkpoint_improvements.py`, and insert the following code:

```
1 # import the necessary packages
2 from sklearn.preprocessing import LabelBinarizer
3 from pyimagesearch.nn.conv import MiniVGGNet
4 from keras.callbacks import ModelCheckpoint
5 from keras.optimizers import SGD
6 from keras.datasets import cifar10
7 import argparse
8 import os
```

Lines 2-8 import our required Python packages. Take note of the `ModelCheckpoint` class imported on **Line 4** – this class will enable us to checkpoint and serialize our networks to disk whenever we find an incremental improvement in model performance.

Next, let's parse our command line arguments:

```
10 # construct the argument parse and parse the arguments
11 ap = argparse.ArgumentParser()
12 ap.add_argument("-w", "--weights", required=True,
13     help="path to weights directory")
14 args = vars(ap.parse_args())
```

The only command line argument we need is `--weights`, the path to the output directory that will store our serialized models during the training process. We then perform our standard routine of loading the CIFAR-10 dataset from disk, scaling the pixel intensities to the range $[0, 1]$, and then one-hot encoding the labels:

```
16 # load the training and testing data, then scale it into the
17 # range [0, 1]
18 print("[INFO] loading CIFAR-10 data...")
19 ((trainX, trainY), (testX, testY)) = cifar10.load_data()
20 trainX = trainX.astype("float") / 255.0
21 testX = testX.astype("float") / 255.0
22
23 # convert the labels from integers to vectors
24 lb = LabelBinarizer()
25 trainY = lb.fit_transform(trainY)
26 testY = lb.transform(testY)
```

Given our data, we are now ready to initialize our SGD optimizer along with the MiniVGGNet architecture:

```
28 # initialize the optimizer and model
29 print("[INFO] compiling model...")
30 opt = SGD(lr=0.01, decay=0.01 / 40, momentum=0.9, nesterov=True)
31 model = MiniVGGNet.build(width=32, height=32, depth=3, classes=10)
32 model.compile(loss="categorical_crossentropy", optimizer=opt,
33     metrics=["accuracy"])
```

We'll use the SGD optimizer with an initial learning rate of $\alpha = 0.01$ and then slowly decay it over the course of 40 epochs. We'll also apply a momentum of $\gamma = 0.9$ and indicate that Nesterov acceleration should also be used as well.

The MiniVGGNet architecture is instantiated to accept input images with a width of 32 pixels, a height of 32 pixels, and a depth of 3 (number of channels). We set `classes=10` since the CIFAR-10 dataset has ten possible class labels.

The critical step to checkpointing our network can be found in the code block below:

```
35 # construct the callback to save only the *best* model to disk
36 # based on the validation loss
37 fname = os.path.sep.join([args["weights"],
38     "weights-{epoch:03d}-{val_loss:.4f}.hdf5"])
```

```
39 checkpoint = ModelCheckpoint(fname, monitor="val_loss", mode="min",
40     save_best_only=True, verbose=1)
41 callbacks = [checkpoint]
```

On **Lines 37 and 38** we construct a special filename (`fname`) template string that Keras uses when writing our models to disk. The first variable in the template, `{epoch:03d}`, is our epoch number, written out to three digits.

The second variable is the metric we want to monitor for improvement, `{val_loss:.4f}`, the loss itself for validation set on the current epoch. Of course, if we wanted to monitor the validation accuracy we can replace `val_loss` with `val_acc`. If we instead wanted to monitor the *training* loss and accuracy the variable would become `train_loss` and `train_acc`, respectively (although I would recommend *monitoring your validation metrics* as they will give you a better sense on how your model will generalize).

Once the output filename template is defined, we then instantiate the `ModelCheckpoint` class on **Lines 39 and 40**. The first parameter to `ModelCheckpoint` is the string representing our filename template. We then pass in what we would like to `monitor`. In this case, we would like to monitor the validation loss (`val_loss`).

The `mode` parameter controls whether the `ModelCheckpoint` should be looking for values that *minimize* our metric or *maximize it*. Since we are working with loss, lower is better, so we set `mode="min"`. If we were instead working with `val_acc`, we would set `mode="max"` (since higher accuracy is better).

Setting `save_best_only=True` ensures that the latest best model (according to the metric monitored) will not be overwritten. Finally, the `verbose=1` setting simply logs a notification to our terminal when a model is being serialized to disk during training.

Line 41 then constructs a list of `callbacks` – the only callback we need is our `checkpoint`.

The last step is to simply train the network and allowing our `checkpoint` to take care of the rest:

```
43 # train the network
44 print("[INFO] training network...")
45 H = model.fit(trainX, trainY, validation_data=(testX, testY),
46     batch_size=64, epochs=40, callbacks=callbacks, verbose=2)
```

To execute our script, simply open up a terminal and execute the following command:

```
$ python cifar10_checkpoint_improvements.py --weights weights/improvements
[INFO] loading CIFAR-10 data...
[INFO] compiling model...
[INFO] training network...
Train on 50000 samples, validate on 10000 samples
Epoch 1/40
171s - loss: 1.6700 - acc: 0.4375 - val_loss: 1.2697 - val_acc: 0.5425
Epoch 2/40
Epoch 00001: val_loss improved from 1.26973 to 0.98481, saving model to test/
    weights-001-0.9848.hdf5
...
Epoch 40/40
Epoch 00039: val_loss did not improve
315s - loss: 0.2594 - acc: 0.9075 - val_loss: 0.5707 - val_acc: 0.8190
```

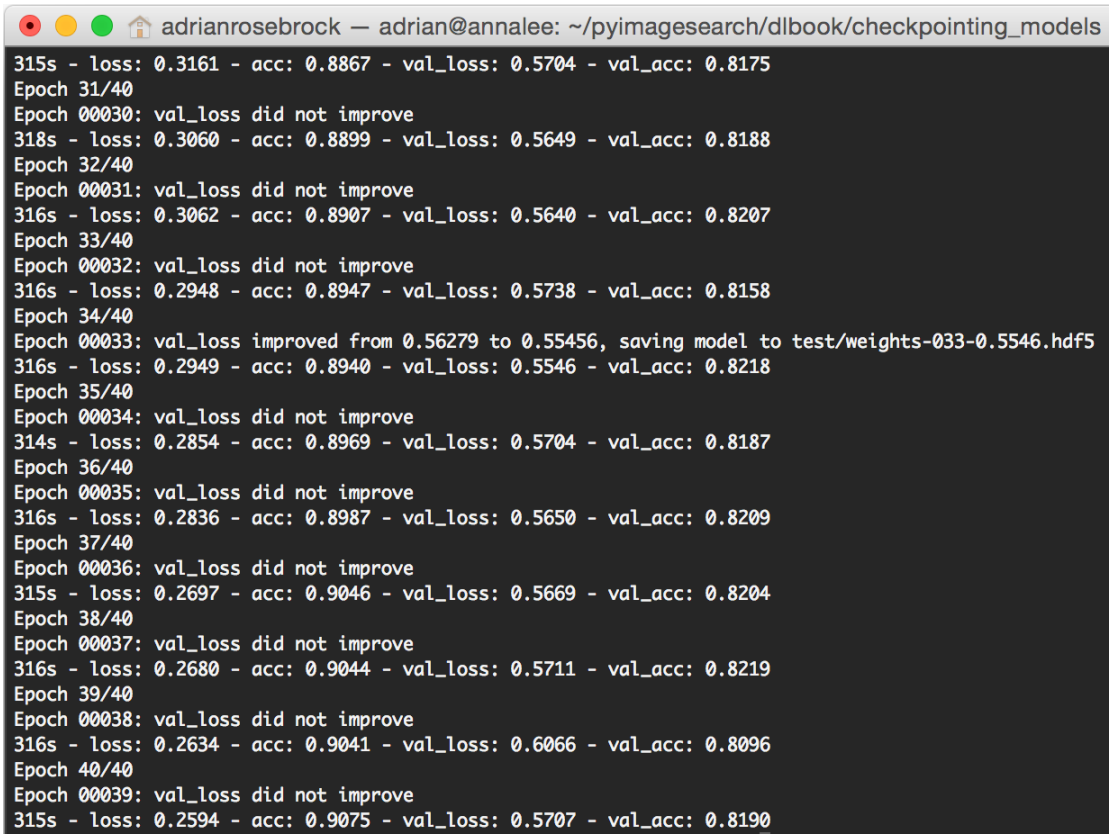

Figure 18.1: Checkpointing individual models every time model performance *improves*, resulting in *multiple weight files* after training completes.

As we can see from my terminal output and Figure 18.1, every time the validation loss decreases we save a new serialized model to disk.

At the end of the training process we have 18 separate files, one for each incremental improvement:

```
$ find ./  -printf "%f\n" | sort
./
weights-000-1.2697.hdf5
weights-001-0.9848.hdf5
weights-003-0.8176.hdf5
weights-004-0.7987.hdf5
weights-005-0.7722.hdf5
weights-006-0.6925.hdf5
weights-007-0.6846.hdf5
weights-008-0.6771.hdf5
weights-009-0.6212.hdf5
weights-012-0.6121.hdf5
weights-013-0.6101.hdf5
weights-014-0.5899.hdf5
weights-015-0.5811.hdf5
weights-017-0.5774.hdf5
weights-019-0.5740.hdf5
weights-022-0.5724.hdf5
```

```
weights-024-0.5628.hdf5
weights-033-0.5546.hdf5
```

As you can see, each filename has three components. The first is a static string, *weights*. We then have the *epoch number*. The final component of the filename is the metric we are measuring for improvement, which in this case is *validation loss*.

Our best validation loss was obtained on epoch 33 with a value of 0.5546. We could then take this model and load it from disk (Chapter 13) and further evaluate it or apply it to our own custom images (which we'll cover in the next chapter).

Keep in mind that your results will not match mine as networks are stochastic and initialized with random variables. Depending on the initial values, you might have dramatically different model checkpoints, but at the end of the training process, our networks should obtain similar accuracy ($\pm$ a few percentage points).

18.2 Checkpointing Best Neural Network Only

Perhaps the biggest downside with checkpointing incremental improvements is that we end up with a *bunch* of extra files that we are (unlikely) interested in, which is especially true if our validation loss moves up and down over training epochs – each of these incremental improvements will be captured and serialized to disk. In this case, it's best to save only *one* model and simply *overwrite it* every time our metric improves during training.

Luckily, accomplishing this action is as simple as updating the `ModelCheckpoint` class to accept a *simple string* (i.e., a file path *without* any template variables). Then, whenever our metric improves, that file is simply overwritten. To understand the process, let's create a second Python file named `cifar10_checkpoint_best.py` and review the differences.

First, we need to import our required Python packages:

```
# import the necessary packages
from sklearn.preprocessing import LabelBinarizer
from pyimagesearch.nn.conv import MiniVGGNet
from keras.callbacks import ModelCheckpoint
from keras.optimizers import SGD
from keras.datasets import cifar10
import argparse
```

Then parse our command line arguments:

```
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-w", "--weights", required=True,
	help="path to best model weights file")
args = vars(ap.parse_args())
```

The *name* of the command line argument itself is the same (`--weights`), but the *description* of the switch is now different: "path to *best* model weights *file*". Thus, this command line argument will be a simple string to an output path – there will be no tempting applied to this string.

From there we can load our CIFAR-10 dataset and prepare it for training:

```
# 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()
trainY = lb.fit_transform(trainY)
testY = lb.transform(testY)
```

As well as initialize our SGD optimizer and MiniVGGNet architecture:

```
# initialize the optimizer and model
print("[INFO] compiling model...")
opt = SGD(lr=0.01, decay=0.01 / 40, momentum=0.9, nesterov=True)
model = MiniVGGNet.build(width=32, height=32, depth=3, classes=10)
model.compile(loss="categorical_crossentropy", optimizer=opt,
    metrics=["accuracy"])
```

We are now ready to update the `ModelCheckpoint` code:

```
# construct the callback to save only the *best* model to disk
# based on the validation loss
checkpoint = ModelCheckpoint(args["weights"], monitor="val_loss",
    save_best_only=True, verbose=1)
callbacks = [checkpoint]
```

Notice how the `fname` template string is gone – all we are doing is supply the value of `--weights` to `ModelCheckpoint`. Since there are no template values to fill in, Keras will simply *overwrite* the existing serialized weights file whenever our monitoring metric improves (in this case, validation loss).

Finally, we train on network in the code block below:

```
# train the network
print("[INFO] training network...")
H = model.fit(trainX, trainY, validation_data=(testX, testY),
    batch_size=64, epochs=40, callbacks=callbacks, verbose=2)
```

To execute our script, issue the following command:

```
$ python cifar10_checkpoint_best.py --weights test_best/cifar10_best_weights.hdf5
[INFO] loading CIFAR-10 data...
[INFO] compiling model...
[INFO] training network...
Train on 50000 samples, validate on 10000 samples
Epoch 1/40
Epoch 00000: val_loss improved from inf to 1.26677, saving model to
    test_best/cifar10_best_weights.hdf5
```

```
305s - loss: 1.6657 - acc: 0.4441 - val_loss: 1.2668 - val_acc: 0.5584
Epoch 2/40
Epoch 00001: val_loss improved from 1.26677 to 1.21923, saving model to
     test_best/cifar10_best_weights.hdf5
309s - loss: 1.1996 - acc: 0.5828 - val_loss: 1.2192 - val_acc: 0.5798
...
Epoch 40/40
Epoch 00039: val_loss did not improve
173s - loss: 0.2615 - acc: 0.9079 - val_loss: 0.5511 - val_acc: 0.8250
```

Here you can see that we overwrite our `cifar10_best_weights.hdf5` file with the updated network *only* if our validation loss decreases. This has two primary benefits:

1. There is only *one* serialized file at the end of the training process – the model epoch that obtained the lowest loss.
2. We are not capturing "incremental improvements" where loss fluctuates up and down. Instead, we only save and overwrite the *existing* best model if our metric obtains a loss lower than *all* previous epochs.

To confirm this, take a look at my `weights/best` directory where you can see there is only one output file:

```
$ ls -l weights/best/
total 17024
-rw-rw-r-- 1 adrian adrian 17431968 Apr 28 09:47 cifar10_best_weights.hdf5
```

You can then take this serialized MiniVGGNet and further evaluate it on the testing data or apply it to your own images (covered in the Chapter 15).

18.3 Summary

In this chapter, we reviewed how to monitor a given metric (e.x., validation loss, validation accuracy, etc.) during training and then save high performing networks to disk. There are two methods to accomplish this inside Keras:

1. Checkpoint *incremental* improvements.
2. Checkpoint *only* the best model found during the process.

Personally, I prefer the latter over the former since it results in less files and a single output file that represents the best epoch found during the training process.

19. Visualizing Network Architectures

One concept we have not discussed yet is *architecture visualization*, the process of constructing a *graph* of nodes and associated connections in a network and saving the graph to disk as an image (i.e., .PNG. JPG, etc.). Nodes in the graphs represent layers, while connections between nodes represent the flow of data through the network.

These graphs typically include the following components for each layer:

1. The *input* volume size.
2. The *output* volume size.
3. And optionally the *name* of the layer.

We typically use network architecture visualization when (1) debugging our own custom network architectures and (2) publication, where a visualization of the architecture is easier to understand than including the actual source code or trying to construct a table to convey the same information. In the remainder of this chapter, you will learn how to construct network architecture visualization graphs using Keras, followed by serializing the graph to disk as an actual image.

19.1 The Importance of Architecture Visualization

Visualizing the architecture of a model is a critical debugging tool, *especially* if you are:

1. Implementing an architecture in a publication, but are unfamiliar with it.
2. Implementing your own custom network architecture.

In short, network visualization *validates our assumptions* that our code is correctly building the model we are intending to construct. By examining the output graph image, you can see if there is a flaw in your logic. The most common flaws include:

1. Incorrectly ordering layers in the network.
2. Assuming an (incorrect) output volume size after a `CONV` or `POOL` layer.

Whenever implementing a network architecture, I suggest you visualize the network architecture after every block of `CONV` and `POOL` layers, which will enable you to validate your assumptions (and more importantly, catch "bugs" in the network early on).

Bugs in Convolutional Neural Networks are not like other logic bugs in applications resulting from edge cases. Instead, a CNN very well may train and obtain reasonable results even with an

incorrect layer ordering, but if you don't *realize* that this bug has happened, you might report your results thinking you did one thing, but in reality did another.

In the remainder of this chapter, I'll help you visualize your own network architectures to avoid these types of problematic situations.

19.1.1 Installing graphviz and pydot

In order to construct a graph of our network and save it to disk using Keras, we need to install the `graphviz` prerequisite:

On Ubuntu, this is as simple as:

```
$ sudo apt-get install graphviz
```

While on macOS, we can install `graphviz` via Homebrew:

```
$ brew install graphviz
```

Once `graphviz` library is installed, we need to install two Python packages:

```
$ pip install graphviz==0.5.2
$ pip install pydot-ng==1.0.0
```

The above instructions were included in Chapter 6 when you configured your development machine for deep learning, but I've included them here as well as a matter of completeness. If you are struggling to get these libraries installed, please see the associated supplementary material at the end of Chapter 6.

19.1.2 Visualizing Keras Networks

Visualizing network architectures with Keras is incredibly simple. To see how easy it is, open up a new file, name it `visualize_architecture.py` and insert the following code:

```
1 # import the necessary packages
2 from pyimagesearch.nn.conv import LeNet
3 from keras.utils import plot_model
4
5 # initialize LeNet and then write the network architecture
6 # visualization graph to disk
7 model = LeNet.build(28, 28, 1, 10)
8 plot_model(model, to_file="lenet.png", show_shapes=True)
```

Line 2 imports our implementation of LeNet (Chapter 14) – this is the network architecture that we'll be visualizing. **Line 3** imports the `plot_model` function from Keras. As this function name suggests, `plot_model` is responsible for constructing a graph based on the layers inside the input model and then writing the graph to disk an image.

On **Line 7** we instantiate the LeNet architecture as if we were going to apply it to MNIST for digit classification. The parameters include the width of the input volume (28 pixels), the height (28 pixels), the depth (1 channel), and the total number of class labels (10).

Finally, **Line 8** plots our `model` saves it to disk under the name `lenet.png`.

To execute our script, just open up a terminal and issue the following command:

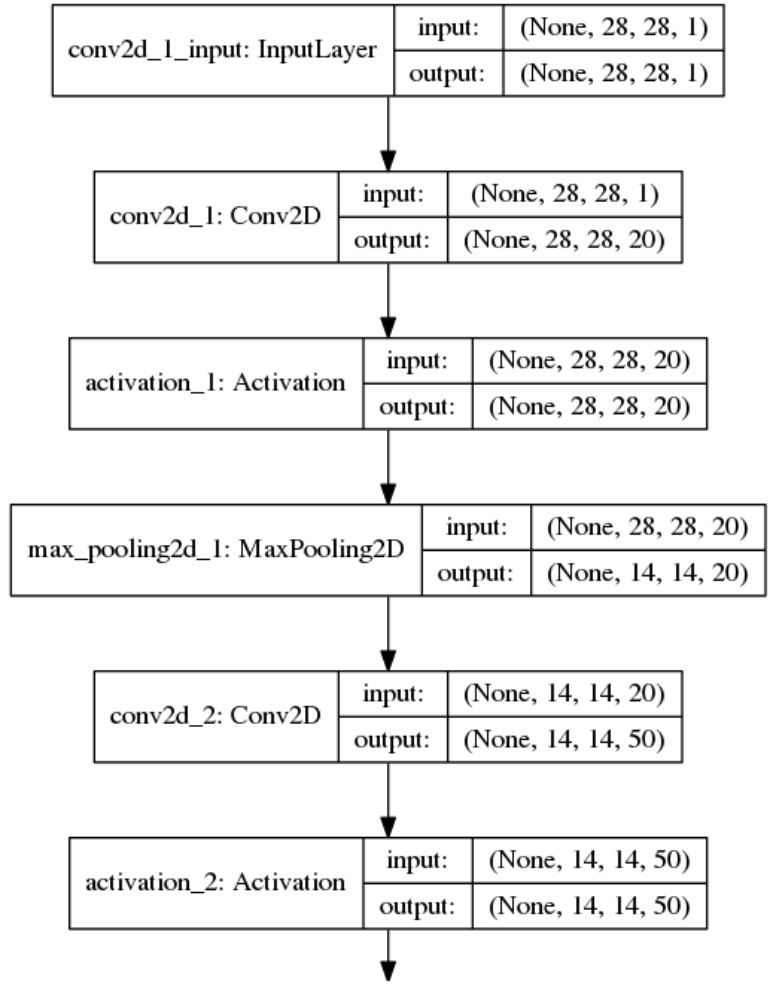

Figure 19.1: Part I of a graphical depiction of the LeNet network architecture generated by Keras. Each node in the graph represents a specific layer function (i.e., convolution, pooling, activation, flattening, fully-connected, etc.). Arrows represent the flow of data through the network. Each node also includes the volume input size and output size after a given operation.

```
$ python visualize_architecture.py
```

Once the command successfully exists, check your current working directory:

```
$ ls
lenet.png  visualize_architecture.py
```

As you'll see, there is a file named `lenet.png` – this file is our actual network visualization graph. Open it up and examine it (Figures 19.1 and 19.2).

Here we can see a visualization of the data flow through our network. Each layer is represented as a node in the architecture which are then connected to other layers, ultimately terminating after the softmax classifier is applied. Notice how each layer in the network includes an `input` and `output` attribute – these values are the size of the respective volume's spatial dimensions when it *enters* the layer and after it *exits* the layer.

Walking through the LeNet architecture, we see the first layer is our `InputLayer` which accepts a $28 \times 28 \times 1$ input image. The spatial dimensions for the input and output of the layer are the same as this is simply a "placeholder" for the input data.

You might be wondering what the `None` represents in the data shape `(None, 28, 28, 1)`. The `None` is actually our batch size. When visualizing the network architecture, Keras does not know our intended batch size so it leaves the value as `None`. When training this value would change to 32, 64, 128, etc., or whatever batch size we deemed appropriate.

Next, our data flows to the first `CONV` layer, where we learn 20 kernels on the $28 \times 28 \times 1$ input. The output of this first `CONV` layer is $28 \times 28 \times 20$. We have retained our original spatial dimensions due to zero padding, but by learning 20 filters we have changed the volume size.

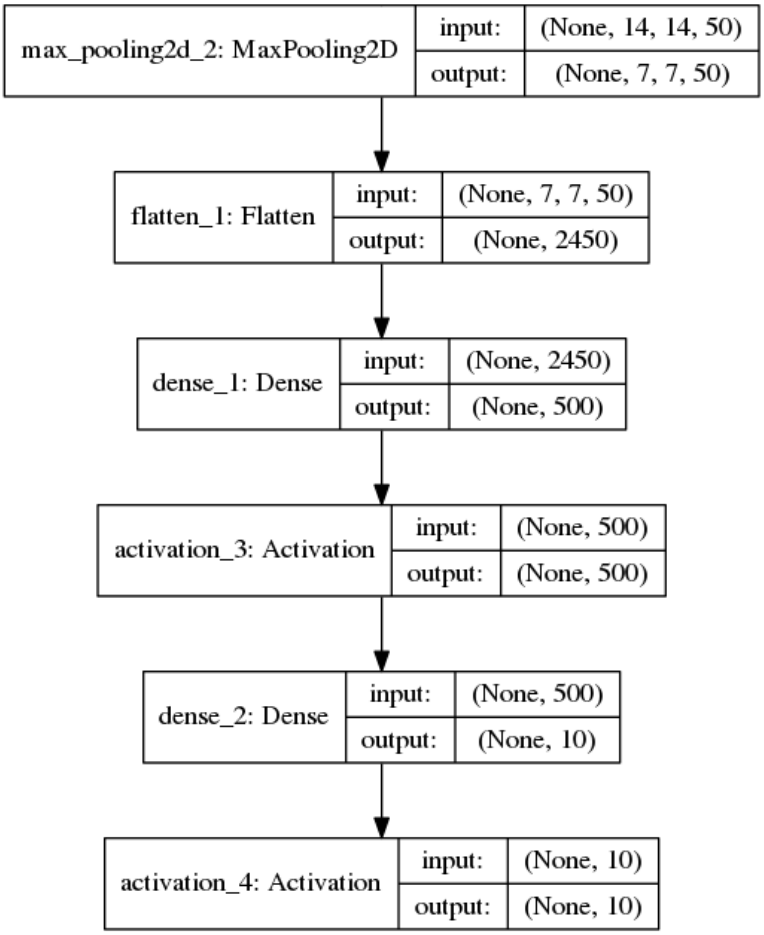

Figure 19.2: Part II of the LeNet architecture visualization, including the fully-connected layers and softmax classifier. In this case, we assume our instantiation of LeNet will be used with the MNIST dataset so we have ten total output nodes in our final softmax layer.

An activation layer follows the `CONV` layer, which by definition cannot change the input volume size. However, a `POOL` operation *can* reduce the volume size – here our input volume is reduced from $28 \times 28 \times 20$ down to $14 \times 14 \times 20$.

The second `CONV` accepts the $14 \times 14 \times 20$ volume as input, but then learns 50 filters, changing the output volume size to $14 \times 14 \times 50$ (again, zero padding is leveraged to ensure the convolution itself does not reduce the width and height of the input). An activation is applied prior to another

POOL operation which again halves the width and height from $14 \times 14 \times 50$ down to $7 \times 7 \times 50$.

At this point, we are ready to apply our FC layers. To accomplish this, our $7 \times 7 \times 50$ input is flattened into a list of *2,450* values (since $7 \times 7 \times 50 = 2,450$). Now that we have flattened the output of the convolutional part of our network, we can apply a FC layer that accepts the 2,450 input values and learns 500 nodes. An activation follows, followed by another FC layer, this time reducing 500 down to 10 (the total number of class labels for the MNIST dataset).

Finally, a softmax classifier is applied to each of the 10 input nodes, giving us our final class probabilities.

19.2 Summary

Just as we can express the LeNet architecture in code, we can also visualize the model itself as an image. As you get started on your deep learning journey, I *highly encourage* you to use this code to visualize any networks you are working with, *especially* if you are unfamiliar with them. Ensuring you understand the flow of data through the network and how the volume sizes change based on CONV, POOL, and FC layers will give you a dramatically more intimate understanding of the architecture rather than relying on code alone.

When implementing my own network architectures, I validate that I'm on the right track by visualizing the architecture every 2-3 layer blocks *as I'm actually coding the network* – this action helps me find bugs or flaws in my logic early on.

20. Out-of-the-box CNNs for Classification

Thus far we have learned how to train our own custom Convolutional Neural Networks from scratch. Most of these CNNs have been on the more shallow side (and on smaller datasets) so they can be easily trained on our CPUs, without having to resort to more expensive GPUs, which allows us to master the basics of neural networks and deep learning without having to empty our pockets.

However, because we have been working with more shallow networks and smaller datasets, we haven't been able to take advantage of the full classification power that deep learning affords us. Luckily, the Keras library ships with *five* CNNs that have been *pre-trained* on the ImageNet dataset:

- VGG16
- VGG19
- ResNet50
- Inception V3
- Xception

As we discussed in Chapter 5, the goal of the *ImageNet Large Scale Visual Recognition Challenge (ILSVRC)* [42] is to train a model that can correctly classify an input image into *1,000 separate object categories*. These 1,000 image categories represent object classes that we encounter in our day-to-day lives, such as species of dogs, cats, various household objects, vehicle types, and much more.

This implies that if we leverage CNNs *pre-trained* on the ImageNet dataset, we can recognize ***all of these 1,000 object categories out-of-the-box*** – no training required! A complete list of object categories that you can recognize using pre-trained ImageNet models can be found here http://pyimg.co/x1ler.

In this chapter, we'll review the pre-trained state-of-the-art ImageNet models inside the Keras library. I'll then demonstrate how we can write a Python script to use these networks to classify our own custom images *without* having to train these models from scratch.

20.1 State-of-the-art CNNs in Keras

At this point, you're probably wondering:

"I don't have an expensive GPU. How can I use these massive deep learning networks that have been pre-trained on datasets much larger than what we've worked with in this book?"

To answer that question, consider Chapter 8 on *Parameterized Learning*. Recall that the point of parameterized learning is two-fold:

1. Define a machine learning model that can learn patterns from our input data during training time (requiring us to spend more time on the training process), but have the testing process be much faster.
2. Obtain a model that can be defined using a small number of parameters that can easily represent the network, *regardless of training size*.

Therefore, our actual model size is a function of its *parameters*, not the amount of training data. We could train a very deep CNN (such as VGG or ResNet) on a dataset of 1 million images or a dataset of 100 images – but the resulting output model size will be the *same* because model size is determined by the architecture that we choose.

Secondly, neural networks frontload the vast majority of the work. We spend most of our time actually *training* our CNNs, whether this is due to the depth of the architecture, the amount of training data, or the number of experiments we have to run to tune our hyperparameters.

Optimized hardware such as GPUs enable us to speed up the training process as we need to perform both the *forward pass* and the *backward pass* in the backpropagation algorithm – as we already know, this process is how our network actually learns. However, once the network is trained, we only need to perform the *forward pass* to classify a given input image. The forward pass is *substantially faster*, enabling us to classify input images using deep neural networks on a CPU.

In most cases, the network architectures presented in this chapter won't be able to achieve true real-time performance on a CPU (for that we'll need a GPU) – but that's okay; you'll still be able to use these pre-trained networks in your own applications. If you're interested in learning how to train state-of-the-art Convolutional Neural Networks from scratch on the challenging ImageNet dataset, be sure to refer to the *ImageNet Bundle* of this book where I demonstrate exactly that.

20.1.1 VGG16 and VGG19

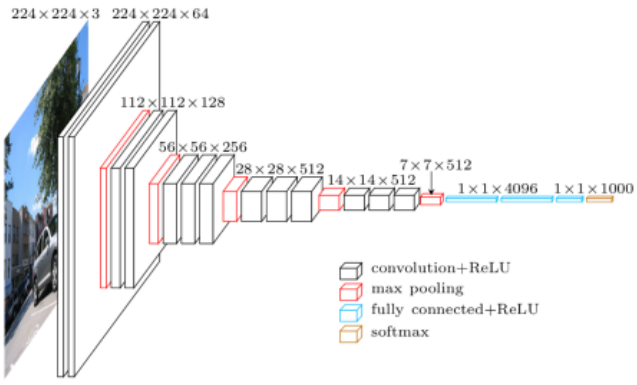

Figure 20.1: A visualization of the VGG architecture. Images with $224 \times 224 \times 3$ dimensions are inputted to the network. Convolution filters of *only* 3×3 are then applied with more convolutions stacked on top of each other prior to max pooling operations deeper in the architecture. *Image credit: http://pyimg.co/xgiek*

The VGG network architecture (Figure 20.1) was introduced by Simonyan and Zisserman in their 2014 paper, *Very Deep Convolutional Networks for Large Scale Image Recognition* [95].

As we discussed in Chapter 15, the VGG family of networks is characterized by using only 3×3 convolutional layers stacked on top of each other in increasing depth. Reducing volume size is handled by max pooling. Two fully-connected layers each with 4,096 nodes are then followed by a softmax classifier.

In 2014, 16 and 19 layer networks were considered *very* deep, although we now have the ResNet architecture which can be successfully trained at depths of 50-200 for ImageNet and over 1,000 for CIFAR-10. Unfortunately, there are two major drawbacks with VGG:

1. It is *painfully slow* to train (luckily we are only testing input images in this chapter).
2. The network weights themselves are quite larger (in terms of disk space/bandwidth). Due to its depth and number of fully-connected nodes, the serialized weight files for VGG16 are 533MB while VGG19 is 574MB.

Luckily, these weights only have to be downloaded *once* – from there we can cache them to disk.

20.1.2 ResNet

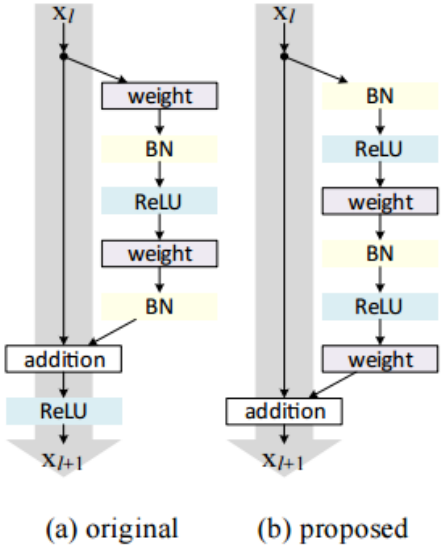

Figure 20.2: **Left:** The original residual module. **Right:** The updated residual module using pre-activation. Figures from He et al., 2016 [130].

First introduced by He et al. in their 2015 paper, *Deep Residual Learning for Image Recognition* [96], the ResNet architecture has become a seminal work in the deep learning literature, demonstrating that *extremely deep* networks can be trained using standard SGD (and a reasonable initialization function) through the use of residual modules.

Further accuracy can be obtained by updating the residual module to use *identity mappings* (Figure 20.2), as demonstrated in their 2016 follow-up publication, *Identity Mappings in Deep Residual Networks* [130].

That said, keep in mind that the ResNet50 (as in 50 weight layers) implementation in the Keras core library is based on the former 2015 paper. Even though ResNet is *much* deeper than both VGG16 and VGG19, the model size is actually *substantially smaller* due to the use of global average pooling rather than fully-connected layers, which reduces the model size down to 102MB for ResNet50.

If you are interested in learning more about the ResNet architecture, including the residual module and how it works, please refer to the *Practitioner Bundle* and *ImageNet Bundle* where ResNet is covered in-depth.

20.1.3 Inception V3

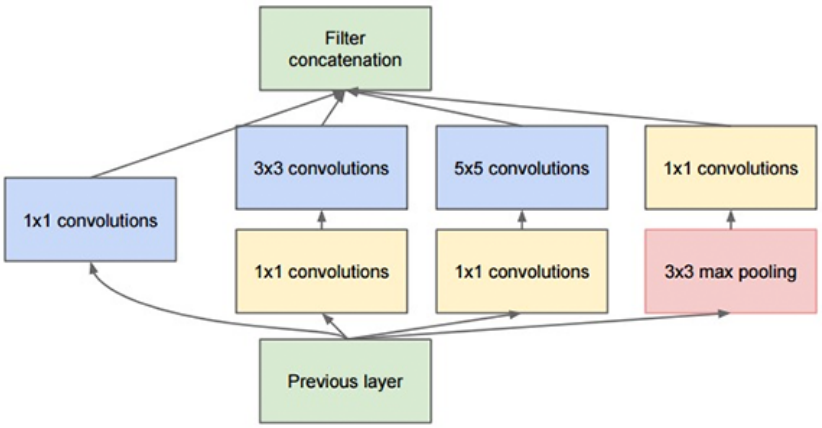

Figure 20.3: The original Inception module used in GoogLeNet. The Inception module acts as a "multi-level feature extractor" by computing 1×1, 3×3, and 5×5 convolutions within the *same* module of the network. Figure from Szegedy et al., 2014 [97].

The "Inception" module (and the resulting Inception architecture) was introduced by Szegedy et al. their 2014 paper, *Going Deeper with Convolutions* [97]. The goal of the inception module (Figure 20.3) is to act as "multi-level feature extractor" by computing 1×1, 3×3, and 5×5 convolutions within the *same* module of the network – the output of these filters are then stacked along the channel dimension before being fed into the next layer in the network.

The original incarnation of this architecture was called *GoogLeNet*, but subsequent manifestations have simply been named *Inception vN* where *N* refers to the version number put out by Google. The Inception V3 architecture included in the Keras core comes from the later by publication by Szegedy et al., *Rethinking the Inception Architecture for Computer Vision* (2015) [131], which proposes updates to the inception module to further boost ImageNet classification accuracy. The weights for Inception V3 are smaller than both VGG and ResNet, coming in at 96MB.

For more information on how the Inception module works (and how to train GoogLeNet from scratch), please refer to the *Practitioner Bundle* and *ImageNet Bundle*.

20.1.4 Xception

Xception was proposed by none other than François Chollet himself, the creator and chief maintainer of the Keras library, in his 2016 paper, *Xception: Deep Learning with Depthwise Separable Convolutions* [132]. Xception is an extension to the Inception architecture which replaces the standard Inception modules with depthwise separable convolutions. The Xception weights are the smallest of the pre-trained networks included in the Keras library, weighing in at 91MB.

20.1.5 Can We Go Smaller?

While it's not included in the Keras library, I wanted to mention that the SqueezeNet architecture [127] is often used when we need a *tiny* footprint. SqueezeNet is *very small* at only ***4.9MB*** and is

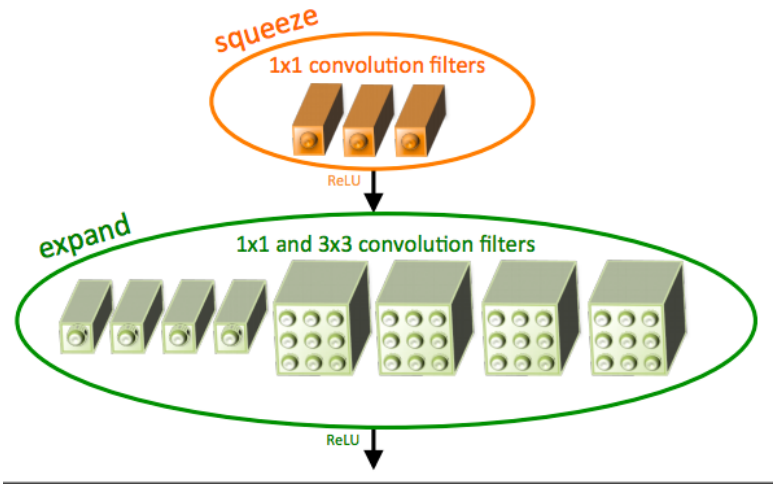

Figure 20.4: The "fire" module in SqueezeNet, consisting of a "squeeze" and "expand". Figure from Iandola et al, 2016 [127].

often used when networks need to be trained and then deployed over a network and/or to resource constrained devices.

Again, SqueezeNet is not included in the Keras core, but I do demonstrate how to train it from scratch on the ImageNet dataset inside the *ImageNet Bundle*.

20.2 Classifying Images with Pre-trained ImageNet CNNs

Let's learn how to classify images with pre-trained Convolutional Neural Networks using the Keras library. We don't have to update our core `pyimagesearch` module that we have been developing thus far as the pre-trained models are already part of the Keras library.

Simply open up a new file, name it `imagenet_pretrained.py`, and insert the following code:

```
1  # import the necessary packages
2  from keras.applications import ResNet50
3  from keras.applications import InceptionV3
4  from keras.applications import Xception # TensorFlow ONLY
5  from keras.applications import VGG16
6  from keras.applications import VGG19
7  from keras.applications import imagenet_utils
8  from keras.applications.inception_v3 import preprocess_input
9  from keras.preprocessing.image import img_to_array
10 from keras.preprocessing.image import load_img
11 import numpy as np
12 import argparse
13 import cv2
```

Lines 2-13 import our required Python packages. As you can see, most of the packages are part of the Keras library. Specifically, **Lines 2-6** handle importing the Keras implementations of ResNet50, Inception V3, Xception, VGG16, and VGG19, respectively. Please note that the Xception network is compatible *only with the TensorFlow backend* (the class will raise an error if you try to instantiate it when using a Theano backend).

Line 7 gives us access to the `imagenet_utils` sub-module, a handy set of convenience functions that will make pre-processing our input images and decoding output classifications easier.

The remainder of the imports are other helper functions, followed by NumPy for numerical operations and `cv2` for our OpenCV bindings.

Next, let's parse our command line arguments:

```
15 # construct the argument parse and parse the arguments
16 ap = argparse.ArgumentParser()
17 ap.add_argument("-i", "--image", required=True,
18      help="path to the input image")
19 ap.add_argument("-model", "--model", type=str, default="vgg16",
20      help="name of pre-trained network to use")
21 args = vars(ap.parse_args())
```

We'll require only a single command line argument, `--image`, which is the path to our input image that we wish to classify. We'll also accept an optional command line argument, `--model`, a string that specifies which pre-trained CNN we would like to use – this value defaults to `vgg16` for the VGG16 architecture.

Given that we accept the name of a pre-trained network via a command line argument, we need to define a Python dictionary that maps the model names (strings) to their actual Keras classes:

```
23 # define a dictionary that maps model names to their classes
24 # inside Keras
25 MODELS = {
26      "vgg16": VGG16,
27      "vgg19": VGG19,
28      "inception": InceptionV3,
29      "xception": Xception,  # TensorFlow ONLY
30      "resnet": ResNet50
31 }
32
33 # ensure a valid model name was supplied via command line argument
34 if args["model"] not in MODELS.keys():
35      raise AssertionError("The --model command line argument should "
36           "be a key in the 'MODELS' dictionary")
```

Lines 25-31 define our `MODELS` dictionary which maps the model name string to the corresponding class. If the `--model` name is not found inside `MODELS`, we'll raise an `AssertionError` (**Lines 34-36**).

As we already know, a CNN takes an image as an input and then returns a set of probabilities corresponding to the class labels as output. Typical input image sizes to a CNN trained on ImageNet are 224×224, 227×227, 256×256, and 299×299; however, you may see other dimensions as well.

VGG16, VGG19, and ResNet all accept 224×224 input images while Inception V3 and Xception require 229×229 pixel inputs, as demonstrated by the following code block:

```
38 # initialize the input image shape (224x224 pixels) along with
39 # the pre-processing function (this might need to be changed
40 # based on which model we use to classify our image)
41 inputShape = (224, 224)
```

```
42 preprocess = imagenet_utils.preprocess_input
43
44 # if we are using the InceptionV3 or Xception networks, then we
45 # need to set the input shape to (299x299) [rather than (224x224)]
46 # and use a different image processing function
47 if args["model"] in ("inception", "xception"):
48      inputShape = (299, 299)
49      preprocess = preprocess_input
```

Here we initialize our `inputShape` to be 224×224 pixels. We also initialize our `preprocess` function to be the standard `preprocess_input` from Keras (which performs mean subtraction, a normalization technique we cover in the *Practitioner Bundle*). However, if we are using Inception or Xception, we need to set the `inputShape` to 299×299 pixels, followed by updating `preprocess` to use a *separate pre-processing function* that performs a *different type of scaling* http://pyimg.co/3ico2.

The next step is to load our pre-trained network architecture weights from disk and instantiate our model:

```
51 # load our the network weights from disk (NOTE: if this is the
52 # first time you are running this script for a given network, the
53 # weights will need to be downloaded first -- depending on which
54 # network you are using, the weights can be 90-575MB, so be
55 # patient; the weights will be cached and subsequent runs of this
56 # script will be *much* faster)
57 print("[INFO] loading {}...".format(args["model"]))
58 Network = MODELS[args["model"]]
59 model = Network(weights="imagenet")
```

Line 58 uses the `MODELS` dictionary along with the `--model` command line argument to grab the correct network class. The CNN is then instantiated on **Line 59** using the pre-trained ImageNet weights.

Again, keep in mind that the weights for VGG16 and VGG19 are 500MB. ResNet weights are $\approx 100MB$, while Inception and Xception weights are between 90-100MB. If this is the *first* time you are running this script for a given network architecture, these weights will be (automatically) downloaded and cached to your local disk. Depending on your internet speed, this may take awhile. However, once the weights are downloaded, they will *not* need to be downloaded again, allowing subsequent runs of `imagenet_pretrained.py` to run ***much faster***.

Our network is now loaded and ready to classify an image – we just need to prepare the image for classification by preprocessing it:

```
61 # load the input image using the Keras helper utility while ensuring
62 # the image is resized to `inputShape`, the required input dimensions
63 # for the ImageNet pre-trained network
64 print("[INFO] loading and pre-processing image...")
65 image = load_img(args["image"], target_size=inputShape)
66 image = img_to_array(image)
67
68 # our input image is now represented as a NumPy array of shape
69 # (inputShape[0], inputShape[1], 3) however we need to expand the
70 # dimension by making the shape (1, inputShape[0], inputShape[1], 3)
71 # so we can pass it through thenetwork
```

```
72  image = np.expand_dims(image, axis=0)
73
74  # pre-process the image using the appropriate function based on the
75  # model that has been loaded (i.e., mean subtraction, scaling, etc.)
76  image = preprocess(image)
```

Line 65 loads our input image from disk using the supplied `inputShape` to resize the width and height of the image. Assuming we are using "channels last" ordering, our input image is now represented as a NumPy array with the shape `(inputShape[0], inputShape[1], 3)`.

However, we train/classify images in *batches* with CNNs, so we need to add an extra dimension to the array via `np.expand_dims` function on **Line 72**. After calling `np.expand_dims`, our `image` will now have the shape `(1, inputShape[0], inputShape[1], 3)`, again, assuming channels last ordering. Forgetting to add this extra dimension will result in an error when you call the `.predict` method of the model.

Lastly, **Line 76** calls the appropriate pre-processing function to perform mean subtraction and/or scaling.

We are now ready to pass our image through the network and obtain the output classifications:

```
78  # classify the image
79  print("[INFO] classifying image with '{}'...".format(args["model"]))
80  preds = model.predict(image)
81  P = imagenet_utils.decode_predictions(preds)
82
83  # loop over the predictions and display the rank-5 predictions +
84  # probabilities to our terminal
85  for (i, (imagenetID, label, prob)) in enumerate(P[0]):
86      print("{}. {}: {:.2f}%".format(i + 1, label, prob * 100))
```

A call to `.predict` on **Line 80** returns the predictions from the CNN. Given these predictions, we pass them into the ImageNet utility function, `.decode_predictions`, to give us a list of ImageNet class label IDs, "human-readable" labels, and the probability associated with each class label. The top-5 predictions (i.e., the labels with the largest probabilities) are then printed to our terminal on **Lines 85 and 86**.

Our final code block will handle loading our image image from disk via OpenCV, drawing the #1 prediction on the image, and finally displaying it to our screen:

```
88  # load the image via OpenCV, draw the top prediction on the image,
89  # and display the image to our screen
90  orig = cv2.imread(args["image"])
91  (imagenetID, label, prob) = P[0][0]
92  cv2.putText(orig, "Label: {}".format(label), (10, 30),
93      cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 255, 0), 2)
94  cv2.imshow("Classification", orig)
95  cv2.waitKey(0)
```

To see our pre-trained ImageNet networks in action, let's move on to the next section.

20.2.1 Classification Results

To classify an image using a pre-trained network and Keras, simply use our `imagenet_pretrained.py` script and then supply (1) a path to your input image that you wish to classify and (2) the name of the network architecture you wish to use.

I have included example commands for each of the available pre-trained networks available in Keras below:

```
$ python imagenet_pretrained.py \
    --image example_images/example_01.jpg --model vgg16
$ python imagenet_pretrained.py \
    --image example_images/example_02.jpg --model vgg19
$ python imagenet_pretrained.py \
    --image example_images/example_03.jpg --model inception
$ python imagenet_pretrained.py \
    --image example_images/example_04.jpg --model xception
$ python imagenet_pretrained.py \
    --image example_images/example_05.jpg --model resnet
```

Figure 20.5 below displays a montage of the results generated for various input images. In each case, the label predicted by the given network architecture accurately reflects the contents of the image.

Figure 20.5: Results of applying various pre-trained ImageNet networks to input images. In each of the examples, the pre-trained network returns correct classifications.

20.3 Summary

In this chapter, we reviewed the five Convolutional Neural Networks pre-trained on the ImageNet dataset inside the Keras library:

1. VGG16
2. VGG19
3. ResNet50
4. Inception V3
5. Xception

We then learned how to use each of these architectures to classify your own input images. Given that the ImageNet dataset consists of 1,000 popular object categories you are likely to encounter in everyday life, these models make for great "general purpose" classifiers. Depending on your own motivation and end goals of studying deep learning, these networks alone may be enough to build your desired application.

However, for readers who are interested in learning more advanced techniques to train *deeper networks* on *larger datasets*, I would absolutely recommend that you read through the *Practitioner Bundle*. For readers who want the *full experience* and discover how to train these state-of-the-art networks on the challenging ImageNet dataset, please refer to the *ImageNet Bundle*.

21. Case Study: Breaking Captchas with a CNN

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

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

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

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

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

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

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

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

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

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

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

21.1 Breaking Captchas with a CNN

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

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

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

21.1.1 A Note on Responsible Disclosure

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

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

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

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

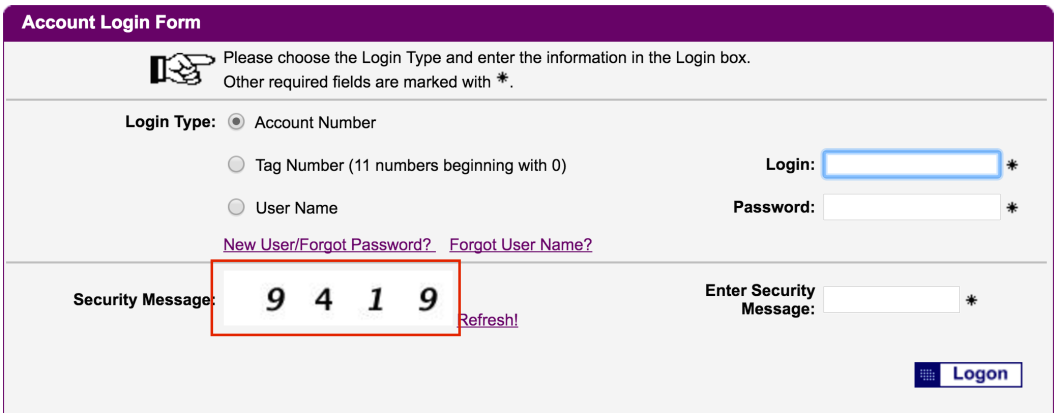

Figure 21.1: The E-Z Pass New York login form. Can you spot the flaw in their login system?

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

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

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

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

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

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

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

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

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

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

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

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

21.1.2 The Captcha Breaker Directory Structure

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

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

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

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

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

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

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

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

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

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

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

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

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

21.1.3 Automatically Downloading Example Images

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

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

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

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

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

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

```
$ pip install requests
```

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

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

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

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

We are now ready to download the captcha images:

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

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

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

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

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

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

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

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

However, 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*).

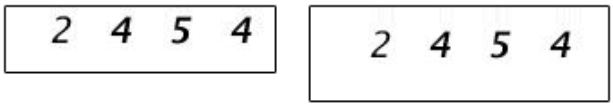

Figure 21.2: **Left:** The original image loaded from disk. **Right:** Padding the image to ensure we can extract the digits *just in case* any of the digits are touching the border of the image.

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 larger enough for us
55              # to see, then wait for a keypress
56              cv2.imshow("ROI", imutils.resize(roi, width=28))
57              key = cv2.waitKey(0)
```

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

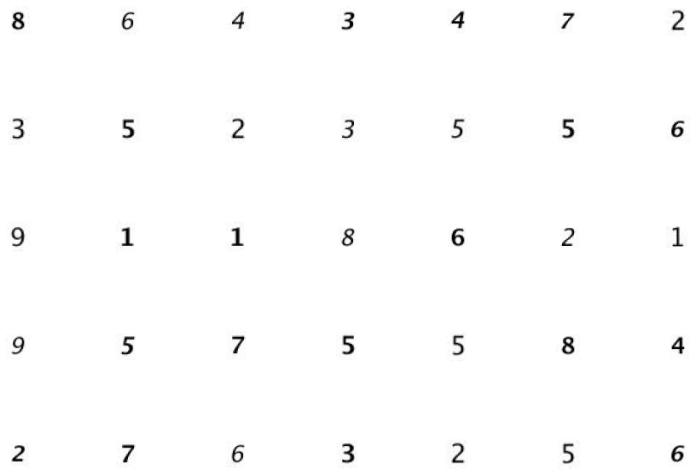

Figure 21.4: A sample of the digit ROIs extracted from our captcha images. Our goal will be to label these images in such a way that we can train a custom Convolutional Neural Network on them.

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

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

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

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

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

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

```
dataset/7
```

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

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

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

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

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

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

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

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

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

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

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

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

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

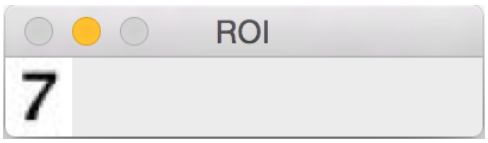

Figure 21.5: When annotating our dataset of digits, a given digit ROI will display on our screen. We then need to press the corresponding key on our keyboard to label the image and save the ROI to disk.

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

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

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

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

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

21.1.5 Preprocessing the Digits

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

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

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

Our `preprocess` function requires three parameters:

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

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

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

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

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

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

21.1.6 Training the Captcha Breaker

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

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

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

Next, let's review our command line arguments:

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

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

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

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

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

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

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

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

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

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

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

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

We can then initialize the LeNet model and SGD optimizer:

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

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

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

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

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

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

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

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

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

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

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

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

avg / total       1.00      1.00      1.00       503

[INFO] serializing network...
```

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

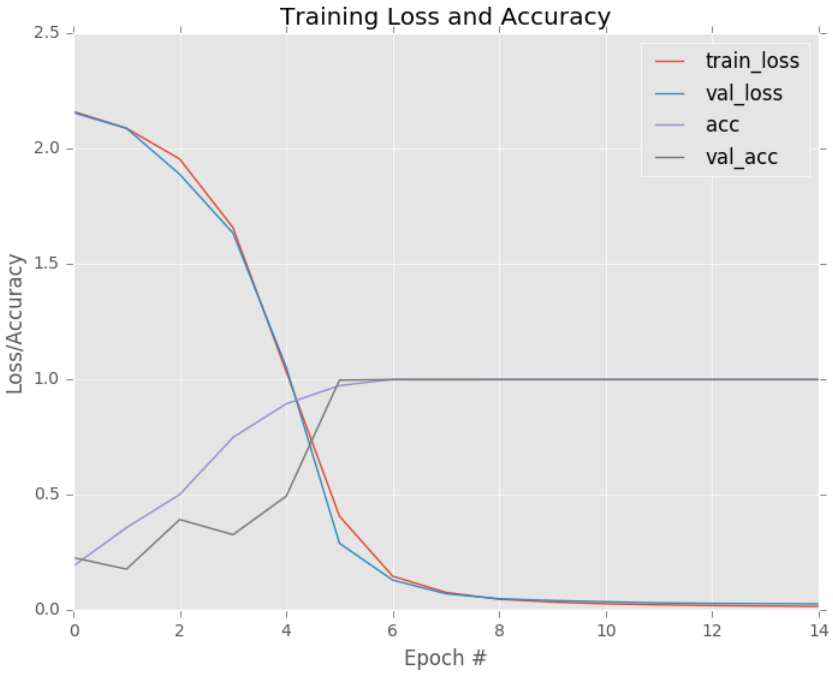

Figure 21.6: Using the LeNet architecture on our custom digits datasets enables us to obtain 100% classification accuracy after only fifteen epochs. Furthermore, there are no signs of overfitting.

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

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

We can then use this model on new input images.

21.1.7 Testing the Captcha Breaker

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

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

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

Next, we'll parse our command line arguments:

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

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

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

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

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

Here comes the fun part – actually breaking the captcha:

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

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

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

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

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

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

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

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

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

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

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

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

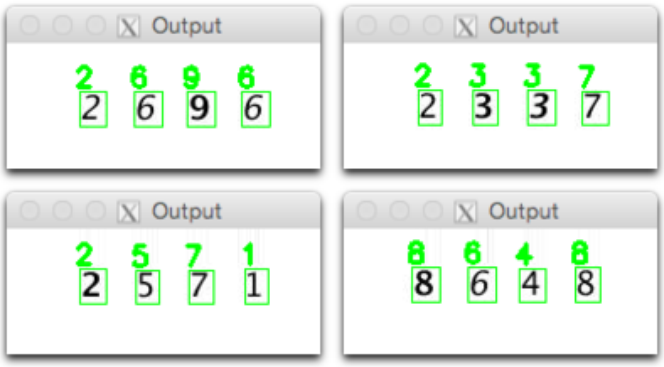

Figure 21.7: Examples of captchas that have been correctly classified and broken by our LeNet model.

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* [8]. The book is a quick read and will give you the foundation you need to be successful when applying deep learning to image classification and computer vision tasks.

22. Case Study: Smile Detection

In this chapter, we will be building a complete end-to-end application that can detect smiles in a video stream in real-time using deep learning along with traditional computer vision techniques.

To accomplish this task, we'll be training the LetNet architecture on a dataset of images that contain faces of people who are *smiling* and *not smiling*. Once our network is trained, we'll create a separate Python script – this one will detect faces in images via OpenCV's built-in Haar cascade face detector, extract the face region of interest (ROI) from the image, and then pass the ROI through LeNet for smile detection.

When developing real-world applications for image classification, you'll often have to mix traditional computer vision and image processing techniques with deep learning. I've done my best to ensure this book stands on its own in terms of algorithms, techniques, and libraries you need to understand in order to be successful when studying and applying deep learning. However, a full review of OpenCV and other computer vision techniques is outside the scope of this book.

To get up to speed with OpenCV and image processing fundamentals, I recommend you read through *Practical Python and OpenCV* – the book is a quick read and will take you less than a weekend to work through. By the time you finish, you'll have a strong understanding of image processing fundamentals.

For a more in-depth treatment of computer vision techniques, be sure to refer to the PyImageSearch Gurus course. Regardless of your background in computer vision and image processing, by the time you have finished this chapter, you'll have a complete smile detection solution that you can use in your own applications.

22.1 The SMILES Dataset

The SMILES dataset consists of images of faces that are either *smiling* or *not smiling* [51]. In total, there are 13,165 grayscale images in the dataset, with each image having a size of 64×64 pixels.

As Figure 22.1 demonstrates, images in this dataset are *tightly cropped* around the face, which will make the training process easier as we'll be able to learn the "smiling" or "not smiling" patterns directly from the input images, just as we have done in similar chapters earlier in this book.

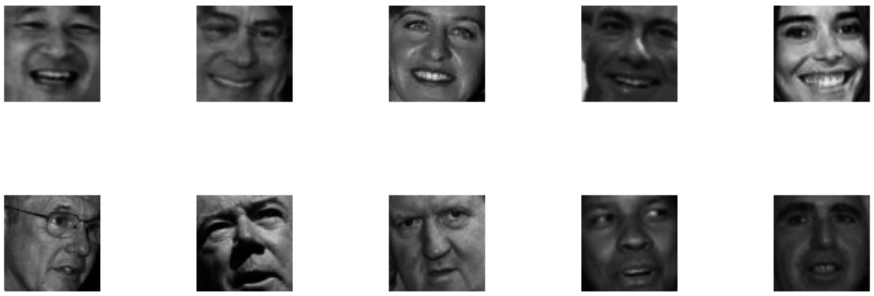

Figure 22.1: Top: Examples of "smiling" faces. Bottom: Samples of "not smiling" faces. In this chapter we will be training a Convolutional Neural Network to recognize between smiling and not smiling faces in real-time video streams.

However, the close cropping poses a problem during testing – since our input images will not only contain a face but the *background* of the image as well, we first need to *localize* the face in the image and extract the face ROI before we can pass it through our network for detection. Luckily, using traditional computer vision methods such as Haar cascades, this is a much easier task than it sounds.

A second issue we need to handle in the SMILES dataset is *class imbalance*. While there are 13,165 images in the dataset, 9,475 of these examples are *not smiling* while only 3,690 belong to the *smiling* class. Given that there are over 2.5x the number of "not smiling" images to "smiling" examples, we need to be careful when devising our training procedure.

Our network may *naturally* pick the "not smiling" label since (1) the distributions are uneven and (2) it has more examples of what a "not smiling" face looks like. As we'll see later in this chapter, we can combat class imbalance by computing a "weight" for each class during training time.

22.2 Training the Smile CNN

The first step in building our smile detector is to train a CNN on the SMILES dataset to distinguish between a face that is smiling versus not smiling. To accomplish this task, let's create a new file named `train_model.py`. From there, insert the following code:

```
# import the necessary packages
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from keras.preprocessing.image import img_to_array
from keras.utils import np_utils
from pyimagesearch.nn.conv import LeNet
from imutils import paths
import matplotlib.pyplot as plt
import numpy as np
import argparse
import imutils
import cv2
import os
```

Lines 2-14 import our required Python packages. We've used all of the packages before, but I want to call your attention to **Line 7** where we import the LeNet (Chapter 14) class – this is the architecture we'll be using when creating our smile detector.

Next, let's parse 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 of faces")
20 ap.add_argument("-m", "--model", required=True,
21      help="path to output model")
22 args = vars(ap.parse_args())
23
24 # initialize the list of data and labels
25 data = []
26 labels = []
```

Our script will require two command line arguments, each of which I've detailed below:

1. `--dataset`: The path to the SMILES directory residing on disk.
2. `--model`: The path to where the serialized LeNet weights will be saved after training.

We are now ready to load the SMILES dataset from disk and store it in memory:

```
28 # loop over the input images
29 for imagePath in sorted(list(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 = imutils.resize(image, width=28)
34      image = img_to_array(image)
35      data.append(image)
36
37      # extract the class label from the image path and update the
38      # labels list
39      label = imagePath.split(os.path.sep)[-3]
40      label = "smiling" if label == "positives" else "not_smiling"
41      labels.append(label)
```

On **Line 29** we loop over all images in the `--dataset` input directory. For each of these images we:

1. Load it from disk (**Line 31**).
2. Convert it to grayscale (**Line 32**).
3. Resize it to have a *fixed input size* of 28×28 pixels (**Line 33**).
4. Convert the image to an array compatible with Keras and its channel ordering (**Line 34**).
5. Add the `image` to the `data` list that LeNet will be trained on.

Lines 39-41 handle extracting the class label from the `imagePath` and updating the `labels` list. The SMILES dataset stores *smiling* faces in the `SMILES/positives/positives7` subdirectory while *not smiling* faces live in the `SMILES/negatives/negatives7` subdirectory.

Therefore, given the path to an image:

```
SMILEs/positives/positives7/10007.jpg
```

We can extract the class label by splitting on the image path separator and grabbing the third-to-last subdirectory: `positives`. In fact, this is exactly what **Line 39** accomplishes.

Now that our `data` and `labels` are constructed, we can scale the raw pixel intensities to the range $[0, 1]$ and then apply one-hot encoding to the `labels`:

```
43 # scale the raw pixel intensities to the range [0, 1]
44 data = np.array(data, dtype="float") / 255.0
45 labels = np.array(labels)
46
47 # convert the labels from integers to vectors
48 le = LabelEncoder().fit(labels)
49 labels = np_utils.to_categorical(le.transform(labels), 2)
```

Our next code block handles our data imbalance issue by computing the class weights:

```
51 # account for skew in the labeled data
52 classTotals = labels.sum(axis=0)
53 classWeight = classTotals.max() / classTotals
```

Line 52 computes the total number of examples per class. In this case, `classTotals` will be an array: `[9475, 3690]` for "not smiling" and "smiling", respectively.

We then *scale* these totals on **Line 53** to obtain the `classWeight` used to handle the class imbalance, yielding the array: `[1, 2.56]`. This weighting implies that our network will treat every instance of "smiling" as 2.56 instances of "not smiling" and helps combat the class imbalance issue by amplifying the per-instance loss by a larger weight when seeing "smiling" examples.

Now that we've computed our class weights, we can move on to partitioning our data into training and testing splits, using 80% of the data for training and 20% for testing:

```
55 # partition the data into training and testing splits using 80% of
56 # the data for training and the remaining 20% for testing
57 (trainX, testX, trainY, testY) = train_test_split(data,
58      labels, test_size=0.20, stratify=labels, random_state=42)
```

Finally, we are ready to train LeNet:

```
60 # initialize the model
61 print("[INFO] compiling model...")
62 model = LeNet.build(width=28, height=28, depth=1, classes=2)
63 model.compile(loss="binary_crossentropy", optimizer="adam",
64      metrics=["accuracy"])
65
66 # train the network
67 print("[INFO] training network...")
68 H = model.fit(trainX, trainY, validation_data=(testX, testY),
69      class_weight=classWeight, batch_size=64, epochs=15, verbose=1)
```

Line 62 initializes the LeNet architecture which will accept 28×28 single channel images. Given that there are only two classes (smiling versus not smiling), we set `classes=2`.

We'll also be using `binary_crossentropy` rather than `categorical_crossentropy` as our loss function. Again, categorical cross-entropy is only used when the number of classes is more than two.

Up until this point, we've been using the SGD optimizer to train our network. Here we'll be using Adam (**Line 63**) [113]. I cover more advanced optimizers (including Adam, RMSprop, Adadelta), and others inside the *Practitioner Bundle*; however, for the sake of this example, simply understand that Adam can converge faster than SGD in certain situations.

Again, the optimizer and associated parameters are often considered hyperparameters that you need to tune when training your network. When I put this example together I found that Adam performed substantially better than SGD.

Lines 68 and 69 train LeNet for a total of 15 epochs using our supplied `classWeight` to combat class imbalance.

Once our network is trained we can evaluate it and serialize the weights to disk:

```
71 # evaluate the network
72 print("[INFO] evaluating network...")
73 predictions = model.predict(testX, batch_size=64)
74 print(classification_report(testY.argmax(axis=1),
75     predictions.argmax(axis=1), target_names=le.classes_))
76
77 # save the model to disk
78 print("[INFO] serializing network...")
79 model.save(args["model"])
```

We'll also construct a learning curve for our network so we can visualize performance:

```
81 # plot the training + testing loss and accuracy
82 plt.style.use("ggplot")
83 plt.figure()
84 plt.plot(np.arange(0, 15), H.history["loss"], label="train_loss")
85 plt.plot(np.arange(0, 15), H.history["val_loss"], label="val_loss")
86 plt.plot(np.arange(0, 15), H.history["acc"], label="acc")
87 plt.plot(np.arange(0, 15), H.history["val_acc"], label="val_acc")
88 plt.title("Training Loss and Accuracy")
89 plt.xlabel("Epoch #")
90 plt.ylabel("Loss/Accuracy")
91 plt.legend()
92 plt.show()
```

To train our smile detector, execute the following command:

```
$ python train_model.py --dataset ../datasets/SMILEsmileD \
    --model output/lenet.hdf5
[INFO] compiling model...
[INFO] training network...
Train on 10532 samples, validate on 2633 samples
Epoch 1/15
8s - loss: 0.3970 - acc: 0.8161 - val_loss: 0.2771 - val_acc: 0.8872
Epoch 2/15
8s - loss: 0.2572 - acc: 0.8919 - val_loss: 0.2620 - val_acc: 0.8899
Epoch 3/15
```

```
7s - loss: 0.2322 - acc: 0.9079 - val_loss: 0.2433 - val_acc: 0.9062
...
Epoch 15/15
8s - loss: 0.0791 - acc: 0.9716 - val_loss: 0.2148 - val_acc: 0.9351
[INFO] evaluating network...
             precision    recall  f1-score   support

not_smiling       0.95      0.97      0.96      1890
    smiling       0.91      0.86      0.88       743

avg / total       0.93      0.94      0.93      2633

[INFO] serializing network...
```

After 15 epochs we can see that our network is obtaining 93% classification accuracy. Figure 22.2 plots our learning curve:

Figure 22.2: A plot of the learning curve for the LeNet architecture trained on the SMILES dataset. After fifteen epochs we are obtaining $\approx 93\%$ classification accuracy on our testing set.

Past epoch six our validation loss starts to stagnate – further training past epoch 15 would result in overfitting. If desired, we would improve the accuracy of our smile detector by using more training data, either by:

1. Gathering additional training data.
2. Applying *data augmentation* to randomly translate, rotate, and shift our *existing* training set.

Data augmentation is covered in detail inside the *Practitioner Bundle*.

22.3 Running the Smile CNN in Real-time

Now that we've trained our model, the next step is to build the Python script to access our webcam/video file and apply smile detection to each frame. To accomplish this step, open up a new file, name it `detect_smile.py`, and we'll get to work.

```
1  # import the necessary packages
2  from keras.preprocessing.image import img_to_array
3  from keras.models import load_model
4  import numpy as np
5  import argparse
6  import imutils
7  import cv2
```

Lines 2-7 import our required Python packages. The `img_to_array` function will be used to convert each individual frame from our video stream to a properly channel ordered array. The `load_model` function will be used to load the weights of our trained LeNet model from disk.

The `detectsmile.py` script requires two command line arguments followed by a third optional one:

```
9  # construct the argument parse and parse the arguments
10 ap = argparse.ArgumentParser()
11 ap.add_argument("-c", "--cascade", required=True,
12      help="path to where the face cascade resides")
13 ap.add_argument("-m", "--model", required=True,
14      help="path to pre-trained smile detector CNN")
15 ap.add_argument("-v", "--video",
16      help="path to the (optional) video file")
17 args = vars(ap.parse_args())
```

The first argument, `--cascade` is the path to a Haar cascade used to detect faces in images. First published in 2001 by Paul Viola and Michael Jones detail the Haar cascade in their work, *Rapid Object Detection using a Boosted Cascade of Simple Features* [134]. This publication has become one of the most cited papers in the computer vision literature.

The Haar cascade algorithm is capable of detecting objects in images, regardless of their location and scale. Perhaps most intriguing (and relevant to our application), the detector can run in real-time on modern hardware. In fact, the motivation of behind Viola and Jones' work was to create a *face detector*.

Because a detailed review of object detection using traditional computer vision methods is outside the scope of this book, you should review of Haar cascades, along with the common Histogram of Oriented Gradients + Linear SVM framework for object detection, by referring to this PyImageSearch blog post (http://pyimg.co/gq9lu) along with the Object Detection module inside PyImageSearch Gurus [33].

The second common line argument, `--model`, specifies the path to our serialized LeNet weights on disk. Our script will *default* to reading frames from a built-in/USB webcam; however, if we instead want to read frames from a file, we can specify the file via the optional `--video` switch.

Before we can detect smiles, we first need to perform some initializations:

```
19 # load the face detector cascade and smile detector CNN
20 detector = cv2.CascadeClassifier(args["cascade"])
```

```
21 model = load_model(args["model"])
22
23 # if a video path was not supplied, grab the reference to the webcam
24 if not args.get("video", False):
25     camera = cv2.VideoCapture(0)
26
27 # otherwise, load the video
28 else:
29     camera = cv2.VideoCapture(args["video"])
```

Lines 20 and 21 load the Haar cascade face detector and the pre-trained LeNet model, respectively. If a video path was *not* supplied, we grab a pointer to our webcam (**Lines 24 and 25**). Otherwise, we open a pointer to the video file on disk (**Lines 28 and 29**).

We have now reached the main processing pipeline of our application:

```
31 # keep looping
32 while True:
33     # grab the current frame
34     (grabbed, frame) = camera.read()
35
36     # if we are viewing a video and we did not grab a frame, then we
37     # have reached the end of the video
38     if args.get("video") and not grabbed:
39         break
40
41     # resize the frame, convert it to grayscale, and then clone the
42     # original frame so we can draw on it later in the program
43     frame = imutils.resize(frame, width=300)
44     gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
45     frameClone = frame.copy()
```

Line 32 starts a loop that will continue until (1) we stop the script or (2) we reach the end of a the video file (provided a `--video` path was applied).

Line 34 grabs the next frame from the video stream. If the `frame` could not be `grabbed`, then we have reached the end of the video file. Otherwise, we pre-process the `frame` for face detection by resizing it to have a width of 300 pixels (**Line 43**) and converting it to grayscale (**Line 44**).

The `.detectMultiScale` method handles detecting the bounding box (x, y)-coordinates of faces in the `frame`:

```
47     # detect faces in the input frame, then clone the frame so that
48     # we can draw on it
49     rects = detector.detectMultiScale(gray, scaleFactor=1.1,
50         minNeighbors=5, minSize=(30, 30),
51         flags=cv2.CASCADE_SCALE_IMAGE)
```

Here we pass in our grayscale image and indicate that for a given region to be considered a face it *must* have a minimum width of 30×30 pixels. The `minNeighbors` attribute helps prune false-positives while the `scaleFactor` controls the number of image pyramid (http://pyimg.co/rtped) levels generated.

Again, a detailed review of Haar cascades for object detection is outside the scope of this book. For a more thorough look at face detection in video streams, please see Chapter 15 of *Practical Python and OpenCV*.

The `.detectMultiScale` method returns a list of 4-tuples that make up the *rectangle* that bounds the face in the `frame`. The first two values in this list are the starting (x, y)-coordinates. The second two values in the `rects` list are the width and height of the bounding box, respectively.

We loop over each set of bounding boxes below:

```
53      # loop over the face bounding boxes
54      for (fX, fY, fW, fH) in rects:
55          # extract the ROI of the face from the grayscale image,
56          # resize it to a fixed 28x28 pixels, and then prepare the
57          # ROI for classification via the CNN
58          roi = gray[fY:fY + fH, fX:fX + fW]
59          roi = cv2.resize(roi, (28, 28))
60          roi = roi.astype("float") / 255.0
61          roi = img_to_array(roi)
62          roi = np.expand_dims(roi, axis=0)
```

For each of the bounding boxes we use NumPy array slicing to extract the face ROI (**Line 58**). Once we have the ROI, we preprocess it and prepare it for classification via LeNet by resizing it, scaling it, converting it to a Keras-compatible array, and padding the image with an extra dimension (**Lines 69-62**).

Once the `roi` is preprocessed, it can be passed through LeNet for classification:

```
64          # determine the probabilities of both "smiling" and "not
65          # smiling", then set the label accordingly
66          (notSmiling, smiling) = model.predict(roi)[0]
67          label = "Smiling" if smiling > notSmiling else "Not Smiling"
```

A call to `.predict` on **Line 66** returns the *probabilities* of "not smiling" and "smiling", respectively. **Line 67** sets the `label` depending on which probability is larger.

Once we have the `label`, we can draw it, along with the corresponding bounding box on the frame:

```
69          # display the label and bounding box rectangle on the output
70          # frame
71          cv2.putText(frameClone, label, (fX, fY - 10),
72              cv2.FONT_HERSHEY_SIMPLEX, 0.45, (0, 0, 255), 2)
73          cv2.rectangle(frameClone, (fX, fY), (fX + fW, fY + fH),
74              (0, 0, 255), 2)
```

Our final code block handles displaying the output frame to our screen:

```
76      # show our detected faces along with smiling/not smiling labels
77      cv2.imshow("Face", frameClone)
78
79      # if the 'q' key is pressed, stop the loop
80      if cv2.waitKey(1) & 0xFF == ord("q"):
81          break
82
83  # cleanup the camera and close any open windows
84  camera.release()
85  cv2.destroyAllWindows()
```

If the q key is pressed, we exit the script.
To run `detect_smile.py` using your webcam, execute the following command:

```
$ python detect_smile.py --cascade haarcascade_frontalface_default.xml \
    --model output/lenet.hdf5
```

If you instead want to use a video file (like I have supplied in the accompanying downloads of this book), you would update your command to use the `--video` switch:

```
$ python detect_smile.py --cascade haarcascade_frontalface_default.xml \
    --model output/lenet.hdf5 --video path/to/your/video.mov
```

I have included the results of the smile detection script in the Figure 22.3 below:

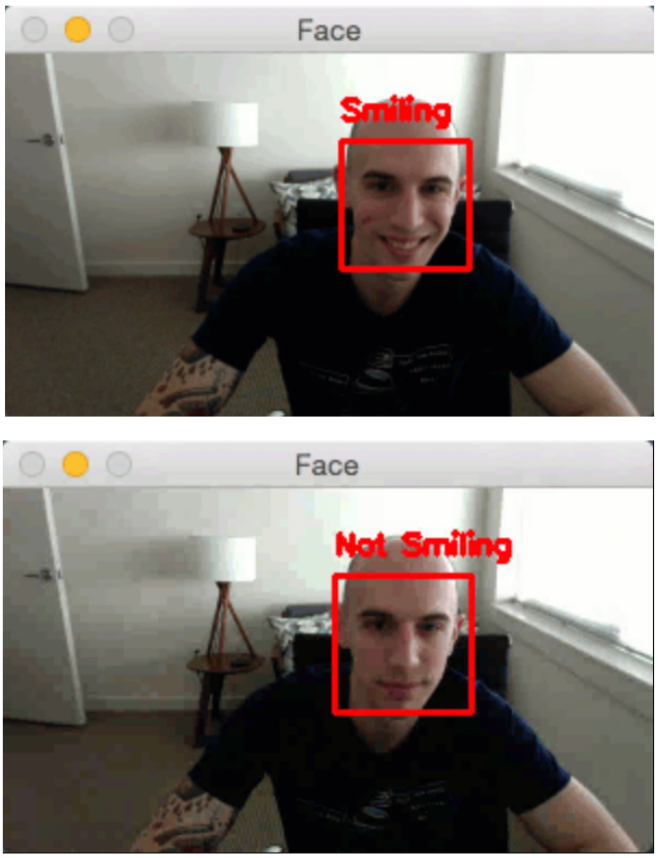

Figure 22.3: Applying our CNN to recognize smiling vs. not-smiling in real-time video streams on a CPU.

Notice how LeNet is correctly predicting "smiling" or "not smiling" based on my facial expression.

22.4 Summary

In this chapter we learned how to build an end-to-end computer vision and deep learning application to perform smile detection. To do so we first trained the LeNet architecture on the SMILES dataset.

Due to class imbalances in the SMILES dataset, we discovered how to compute class weights used to help mitigate the problem.

Once trained, we evaluated LeNet on our testing set and found the network obtained a respectable 93% classification accuracy. Higher classification accuracy can be obtained by gathering more training data or applying data augmentation to *existing* training data.

We then created a Python script to read frames from a webcam/video file, detect faces, and then apply our pre-trained network. In order to detect faces, we used OpenCV's Haar cascades. Once a face was detected it was extracted from the frame and then passed through LeNet to determine if the person was smiling or not smiling. As a whole, our smile detection system can easily run in real-time on the CPU using modern hardware.

23. Your Next Steps

Take a second to congratulate yourself; you've worked through the entire *Starter Bundle* of *Deep Learning for Computer Vision with Python*. That's quite an achievement, and you've earned it.

Let's reflect on your journey. Inside this book you've:

- Learned the fundamentals of image classification.
- Configured your deep learning environment.
- Built your first image classifier.
- Studied parameterized learning.
- Learned all about basic optimization methods (SGD) and regularization techniques.
- Studied Neural Networks inside and out.
- Mastered the fundamentals of Convolutional Neural Networks (CNN).
- Trained your first CNN.
- Investigated more advanced architectures, including LeNet and MiniVGGNet.
- Learned how to spot underfitting and overfitting.
- Applied pre-trained CNNs on the ImageNet dataset to classify to your images.
- Built an end-to-end computer vision system to break captchas.
- Created your own smile detector.

At this point, you have a very strong understanding of the fundamentals of machine learning, neural networks, and deep learning applied to computer vision. But, I have the feeling that your journey is just getting started...

23.1 So, What's Next?

The *Starter Bundle* of *Deep Learning for Computer Vision with Python* is just the tip of the iceberg. This book is meant to help you understand the fundamentals of Convolutional Neural Networks, as well as provide actual end-to-end examples/case studies that you can use to guide you when applying deep learning to your own applications.

But just as deep learning researchers found that *going deeper* leads to more accurate networks, I too would encourage you to take a *deeper dive* into deep learning.

If you want to:

- Understand more advanced training techniques.
- Train your networks faster using transfer-learning.
- Work with large datasets, too big to fit into memory.
- Improve your classification accuracy with network ensembles.
- Explore more exotic architectures such as GoogLeNet and ResNet.
- Study deep dreaming and neural style.
- Learn about Generative Adversarial Networks (GANs).
- Train state-of-the-art architectures such as AlexNet, VGGNet, GoogLeNet, ResNet, and SqueezeNet *from scratch* on the challenging ImageNet dataset...

... then I would highly encourage you to not stop here. Continue your journey towards deep learning mastery. If you enjoyed the *Starter Bundle*, I can guarantee you that the *Practitioner Bundle* and *ImageNet Bundle* only get better from here.

I hope you'll allow me to continue to guide you on your deep learning journey (and avoid the same mistakes I did). If you haven't already picked up a copy of the *Practitioner Bundle* or *ImageNet Bundle*, you can do so here:

https://www.pyimagesearch.com/deep-learning-computer-vision-python-book/

And if you have any questions at all, feel free to contact me:

http://www.pyimagesearch.com/contact/

Cheers,

–Adrian Rosebrock

Bibliography

[1] François Chollet et al. *Keras*. https://github.com/fchollet/keras. 2015 (cited on page 18).

[2] Tianqi Chen et al. "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems". In: *arXiv.org* (Dec. 2015), arXiv:1512.01274. arXiv: 1512.01274 [cs.DC] (cited on page 18).

[3] Martin Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/ (cited on page 18).

[4] Theano Development Team. "Theano: A Python framework for fast computation of mathematical expressions". In: *arXiv e-prints* abs/1605.02688 (May 2016). URL: http://arxiv.org/abs/1605.02688 (cited on page 18).

[5] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pages 2825–2830 (cited on pages 19, 64).

[6] François Chollet. *How does Keras compare to other Deep Learning frameworks like Tensor Flow, Theano, or Torch?* https://www.quora.com/How-does-Keras-compare-to-other-Deep-Learning-frameworks-like-Tensor-Flow-Theano-or-Torch. 2016 (cited on page 19).

[7] Itseez. *Open Source Computer Vision Library (OpenCV)*. https://github.com/itseez/opencv. 2017 (cited on page 19).

[8] Adrian Rosebrock. *Practical Python and OpenCV + Case Studies*. PyImageSearch.com, 2016. URL: https://www.pyimagesearch.com/practical-python-opencv/ (cited on pages 19, 38, 56, 306).

[9] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *Nature* 521.7553 (2015), pages 436–444 (cited on pages 21, 126, 128).

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016 (cited on pages 22, 24, 27, 42, 54, 56, 82, 95, 98, 113, 117, 169, 194, 252).

[11] Warren S. McCulloch and Walter Pitts. "Neurocomputing: Foundations of Research". In: edited by James A. Anderson and Edward Rosenfeld. Cambridge, MA, USA: MIT Press, 1988. Chapter A Logical Calculus of the Ideas Immanent in Nervous Activity, pages 15–27. ISBN: 0-262-01097-6. URL: http://dl.acm.org/citation.cfm?id=65669.104377 (cited on page 22).

[12] F. Rosenblatt. "The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain". In: *Psychological Review* (1958), pages 65–386 (cited on pages 22, 129, 130).

[13] F. Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan, 1962 (cited on page 22).

[14] M. Minsky and S. Papert. *Perceptrons*. Cambridge, MA: MIT Press, 1969 (cited on pages 22, 129).

[15] P. J. Werbos. "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences". PhD thesis. Harvard University, 1974 (cited on pages 23, 129).

[16] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Neurocomputing: Foundations of Research". In: edited by James A. Anderson and Edward Rosenfeld. Cambridge, MA, USA: MIT Press, 1988. Chapter Learning Representations by Back-propagating Errors, pages 696–699. ISBN: 0-262-01097-6. URL: http://dl.acm.org/citation.cfm?id=65669.104451 (cited on pages 23, 129, 137).

[17] Yann LeCun et al. "Efficient BackProp". In: *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*. London, UK, UK: Springer-Verlag, 1998, pages 9–50. ISBN: 3-540-65311-2. URL: http://dl.acm.org/citation.cfm?id=645754.668382 (cited on pages 23, 166).

[18] Balázs Csanád Csáji. "Approximation with Artificial Neural Networks". In: *MSc Thesis, Eötvös Loránd University (ELTE), Budapest, Hungary* (2001) (cited on page 23).

[19] Yann Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE*. 1998, pages 2278–2324 (cited on pages 24, 195, 219, 227).

[20] Jason Brownlee. *What is Deep Learning?* http://machinelearningmastery.com/what-is-deep-learning/. 2016 (cited on page 24).

[21] T. Ojala, M. Pietikainen, and T. Maenpaa. "Multiresolution gray-scale and rotation invariant texture classification with local binary patterns". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 24.7 (2002), pages 971–987 (cited on pages 25, 51, 124).

[22] Robert M. Haralick, K. Shanmugam, and Its'Hak Dinstein. "Textural Features for Image Classification". In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-3.6 (Nov. 1973), pages 610–621. ISSN: 0018-9472. DOI: 10.1109/tsmc.1973.4309314. URL: http://dx.doi.org/10.1109/tsmc.1973.4309314 (cited on page 25).

[23] Ming-Kuei Hu. "Visual pattern recognition by moment invariants". In: *Information Theory, IRE Transactions on* 8.2 (Feb. 1962), pages 179–187. ISSN: 0096-1000 (cited on page 25).

[24] A. Khotanzad and Y. H. Hong. "Invariant Image Recognition by Zernike Moments". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 12.5 (May 1990), pages 489–497. ISSN: 0162-8828. DOI: 10.1109/34.55109. URL: http://dx.doi.org/10.1109/34.55109 (cited on page 25).

[25] Jing Huang et al. “Image Indexing Using Color Correlograms”. In: *Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR ’97)*. CVPR ’97. Washington, DC, USA: IEEE Computer Society, 1997, pages 762–. ISBN: 0-8186-7822-4. URL: http://dl.acm.org/citation.cfm?id=794189.794514 (cited on page 25).

[26] Edward Rosten and Tom Drummond. “Fusing Points and Lines for High Performance Tracking”. In: *Proceedings of the Tenth IEEE International Conference on Computer Vision - Volume 2*. ICCV ’05. Washington, DC, USA: IEEE Computer Society, 2005, pages 1508–1515. ISBN: 0-7695-2334-X-02. DOI: 10.1109/ICCV.2005.104. URL: http://dx.doi.org/10.1109/ICCV.2005.104 (cited on page 25).

[27] Chris Harris and Mike Stephens. “A combined corner and edge detector”. In: *In Proc. of Fourth Alvey Vision Conference*. 1988, pages 147–151 (cited on page 25).

[28] David G. Lowe. “Object Recognition from Local Scale-Invariant Features”. In: *Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2*. ICCV ’99. Washington, DC, USA: IEEE Computer Society, 1999, pages 1150–. ISBN: 0-7695-0164-8. URL: http://dl.acm.org/citation.cfm?id=850924.851523 (cited on page 25).

[29] Herbert Bay et al. “Speeded-Up Robust Features (SURF)”. In: *Comput. Vis. Image Underst.* 110.3 (June 2008), pages 346–359. ISSN: 1077-3142. DOI: 10.1016/j.cviu.2007.09.014. URL: http://dx.doi.org/10.1016/j.cviu.2007.09.014 (cited on page 25).

[30] Michael Calonder et al. “BRIEF: Binary Robust Independent Elementary Features”. In: *Proceedings of the 11th European Conference on Computer Vision: Part IV*. ECCV’10. Heraklion, Crete, Greece: Springer-Verlag, 2010, pages 778–792. ISBN: 3-642-15560-X, 978-3-642-15560-4. URL: http://dl.acm.org/citation.cfm?id=1888089.1888148 (cited on page 25).

[31] Ethan Rublee et al. “ORB: An Efficient Alternative to SIFT or SURF”. In: *Proceedings of the 2011 International Conference on Computer Vision*. ICCV ’11. Washington, DC, USA: IEEE Computer Society, 2011, pages 2564–2571. ISBN: 978-1-4577-1101-5. DOI: 10.1109/ICCV.2011.6126544. URL: http://dx.doi.org/10.1109/ICCV.2011.6126544 (cited on page 25).

[32] Navneet Dalal and Bill Triggs. “Histograms of Oriented Gradients for Human Detection”. In: *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05) - Volume 1 - Volume 01*. CVPR ’05. Washington, DC, USA: IEEE Computer Society, 2005, pages 886–893. ISBN: 0-7695-2372-2. DOI: 10.1109/CVPR.2005.177. URL: http://dx.doi.org/10.1109/CVPR.2005.177 (cited on pages 25, 51, 124).

[33] Adrian Rosebrock. *PyImageSearch Gurus*. https://www.pyimagesearch.com/pyimagesearch-gurus/. 2016 (cited on pages 26, 27, 34, 38, 75, 313).

[34] Pedro F. Felzenszwalb et al. “Object Detection with Discriminatively Trained Part-Based Models”. In: *IEEE Trans. Pattern Anal. Mach. Intell.* 32.9 (Sept. 2010), pages 1627–1645. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2009.167. URL: http://dx.doi.org/10.1109/TPAMI.2009.167 (cited on page 26).

[35] Tomasz Malisiewicz, Abhinav Gupta, and Alexei A. Efros. “Ensemble of Exemplar-SVMs for Object Detection and Beyond”. In: *ICCV*. 2011 (cited on page 26).

[36] Jeff Dean. *Results Get Better With More Data, Larger Models, More Compute*. http://static.googleusercontent.com/media/research.google.com/en//people/jeff/BayLearn2015.pdf. 2016 (cited on page 27).

[37] Geoffrey Hinton. *What Was Actually Wrong With Backpropagation in 1986?* https://www.youtube.com/watch?v=VhmE_UXDOGs. 2016 (cited on page 28).

[38] Andrew Ng. *Deep Learning, Self-Taught Learning and Unsupervised Feature Learning*. https://www.youtube.com/watch?v=n1ViNeWhC24. 2013 (cited on page 29).

[39] Andrew Ng. *What data scientists should know about deep learning*. https://www.slideshare.net/ExtractConf. 2015 (cited on page 29).

[40] Jürgen Schmidhuber. "Deep Learning in Neural Networks: An Overview". In: *CoRR* abs/1404.7828 (2014). URL: http://arxiv.org/abs/1404.7828 (cited on pages 29, 128).

[41] Satya Mallick. *Why does OpenCV use BGR color format ?* http://www.learnopencv.com/why-does-opencv-use-bgr-color-format/. 2015 (cited on page 36).

[42] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pages 211–252. DOI: 10.1007/s11263-015-0816-y (cited on pages 45, 58, 68, 81, 277).

[43] Corinna Cortes and Vladimir Vapnik. "Support-Vector Networks". In: *Mach. Learn.* 20.3 (Sept. 1995), pages 273–297. ISSN: 0885-6125. DOI: 10.1023/A:1022627411411. URL: http://dx.doi.org/10.1023/A:1022627411411 (cited on pages 45, 89).

[44] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. "A Training Algorithm for Optimal Margin Classifiers". In: *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*. COLT '92. Pittsburgh, Pennsylvania, USA: ACM, 1992, pages 144–152. ISBN: 0-89791-497-X. DOI: 10.1145/130385.130401. URL: http://doi.acm.org/10.1145/130385.130401 (cited on page 45).

[45] Leo Breiman. "Random Forests". In: *Mach. Learn.* 45.1 (Oct. 2001), pages 5–32. ISSN: 0885-6125. DOI: 10.1023/A:1010933404324. URL: http://dx.doi.org/10.1023/A:1010933404324 (cited on page 45).

[46] Denny Zhou et al. "Learning with Local and Global Consistency". In: *Advances in Neural Information Processing Systems 16*. Edited by S. Thrun, L. K. Saul, and P. B. Schölkopf. MIT Press, 2004, pages 321–328. URL: http://papers.nips.cc/paper/2506-learning-with-local-and-global-consistency.pdf (cited on page 48).

[47] Xiaojin Zhu and Zoubin Ghahramani. *Learning from Labeled and Unlabeled Data with Label Propagation*. Technical report. 2002 (cited on page 48).

[48] Antti Rasmus et al. "Semi-Supervised Learning with Ladder Network". In: *CoRR* abs/1507.02672 (2015). URL: http://arxiv.org/abs/1507.02672 (cited on page 48).

[49] Avrim Blum and Tom Mitchell. "Combining Labeled and Unlabeled Data with Co-training". In: *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*. COLT' 98. Madison, Wisconsin, USA: ACM, 1998, pages 92–100. ISBN: 1-58113-057-0. DOI: 10.1145/279943.279962. URL: http://doi.acm.org/10.1145/279943.279962 (cited on page 48).

[50] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. *CIFAR-10 and CIFAR-100 (Canadian Institute for Advanced Research)*. http://www.cs.toronto.edu/~kriz/cifar.html (cited on page 55).

[51] Daniel Hromada. *SMILEsmileD*. https://github.com/hromi/SMILEsmileD. 2010 (cited on pages 55, 307).

[52] Maria-Elena Nilsback and Andrew Zisserman. “A Visual Vocabulary for Flower Classification.” In: *CVPR (2)*. IEEE Computer Society, 2006, pages 1447–1454. URL: http://dblp.uni-trier.de/db/conf/cvpr/cvpr2006-2.html#NilsbackZ06 (cited on page 56).

[53] L. Fei-Fei, R. Fergus, and Pietro Perona. “Learning Generative Visual Models From Few Training Examples: An Incremental Bayesian Approach Tested on 101 Object Categories”. In: 2004 (cited on page 57).

[54] Kristen Grauman and Trevor Darrell. “The Pyramid Match Kernel: Efficient Learning with Sets of Features”. In: *J. Mach. Learn. Res.* 8 (May 2007), pages 725–760. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1248659.1248685 (cited on page 57).

[55] Svetlana Lazebnik, Cordelia Schmid, and Jean Ponce. “Beyond Bags of Features: Spatial Pyramid Matching for Recognizing Natural Scene Categories”. In: *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 2*. CVPR ’06. Washington, DC, USA: IEEE Computer Society, 2006, pages 2169–2178. ISBN: 0-7695-2597-0. DOI: 10.1109/CVPR.2006.68. URL: http://dx.doi.org/10.1109/CVPR.2006.68 (cited on page 57).

[56] Hao Zhang et al. “SVM-KNN: Discriminative Nearest Neighbor Classification for Visual Category Recognition”. In: *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 2*. CVPR ’06. Washington, DC, USA: IEEE Computer Society, 2006, pages 2126–2136. ISBN: 0-7695-2597-0. DOI: 10.1109/CVPR.2006.301. URL: http://dx.doi.org/10.1109/CVPR.2006.301 (cited on page 57).

[57] Andrej Karpathy. *CS231n: Convolutional Neural Networks for Visual Recognition*. http://cs231n.stanford.edu/. 2016 (cited on pages 57, 84, 94, 106, 137, 142).

[58] Eran Eidinger, Roee Enbar, and Tal Hassner. “Age and Gender Estimation of Unfiltered Faces”. In: *Trans. Info. For. Sec.* 9.12 (Dec. 2014), pages 2170–2179. ISSN: 1556-6013. DOI: 10.1109/TIFS.2014.2359646. URL: http://dx.doi.org/10.1109/TIFS.2014.2359646 (cited on page 58).

[59] WordNet. *About WordNet*. http://wordnet.princeton.edu. 2010 (cited on page 58).

[60] A. Quattoni and A. Torralba. “Recognizing indoor scenes”. In: *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pages 413–420 (cited on page 60).

[61] Jonathan Krause et al. “3D Object Representations for Fine-Grained Categorization”. In: *4th International IEEE Workshop on 3D Representation and Recognition (3dRR-13)*. Sydney, Australia, 2013 (cited on page 60).

[62] Stéfan van der Walt et al. “scikit-image: image processing in Python”. In: *PeerJ* 2 (June 2014), e453. ISSN: 2167-8359. DOI: 10.7717/peerj.453. URL: http://dx.doi.org/10.7717/peerj.453 (cited on page 64).

[63] Mike Grouchy. *Be Pythonic: __init__.py*. http://mikegrouchy.com/blog/2012/05/be-pythonic-__init__py.html. 2012 (cited on page 69).

[64] Olga Veksler. *k Nearest Neighbors*. http://www.csd.uwo.ca/courses/CS9840a/Lecture2_knn.pdf. 2015 (cited on page 72).

[65] Jon Louis Bentley. “Multidimensional Binary Search Trees Used for Associative Searching”. In: *Commun. ACM* 18.9 (Sept. 1975), pages 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007. URL: http://doi.acm.org/10.1145/361002.361007 (cited on page 79).