DEEP LEARNING FOR COMPUTER VISION

WITH PYTHON

11.3 Common Architectures and Training Patterns 191
11.3.1 Layer Patterns 191
11.3.2 Rules of Thumb 192
11.4 Are CNNs Invariant to Translation, Rotation, and Scaling? 194
11.5 Summary 195

12.1 Keras Configurations and Converting Images to Arrays 197
12.1.1 Understanding the keras.json Configuration File 197
12.1.2 The Image to Array Preprocessor 198
12.2 ShallowNet 200
12.2.1 Implementing ShallowNet 200
12.2.2 ShallowNet on Animals 202
12.2.3 ShallowNet on CIFAR-10 206
12.3 Summary 209

13.1 Serializing a Model to Disk 211
13.2 Loading a Pre-trained Model from Disk 214
13.3 Summary 217

14.1 The LeNet Architecture 219
14.2 Implementing LeNet 220
14.3 LeNet on MNIST 222
14.4 Summary 227

15.1 The VGG Family of Networks 229
15.1.1 The (Mini) VGGNet Architecture 230
15.2 Implementing MiniVGGNet 230
15.3 MiniVGGNet on CIFAR-10 234
15.3.1 With Batch Normalization 236
15.3.2 Without Batch Normalization 237
15.4 Summary 238

16.1 Dropping Our Learning Rate 241
16.1.1 The Standard Decay Schedule in Keras 242
16.1.2 Step-based Decay 243
16.1.3 Implementing Custom Learning Rate Schedules in Keras 244
16.2 Summary 249

column of ones into our design matrix, as long as it exists. Doing so allows us to rewrite our scoring function via a single matrix multiply:

$$f(x_i, W) = Wx_i \tag{9.3}$$

Again, we are allowed to omit the b term here as it is *embedded* into our weight matrix.

In the context of our previous examples in the "Animals" dataset, we've worked with $32 \times 32 \times 3$ images with a total of 3,072 pixels. Each x_i is represented by a vector of $|3072 \times 1|$. Adding in a dimension with a constant value of one now expands the vector to be $|3073 \times 1|$. Similarly, combining both the bias and weight matrix also expands our weight matrix $\boldsymbol{W}$ to be $|3 \times 3073|$ rather than $|3 \times 3072|$. In this way, we can treat the bias as a *learnable parameter within the weight matrix* that we don't have to explicitly keep track of in a separate variable.

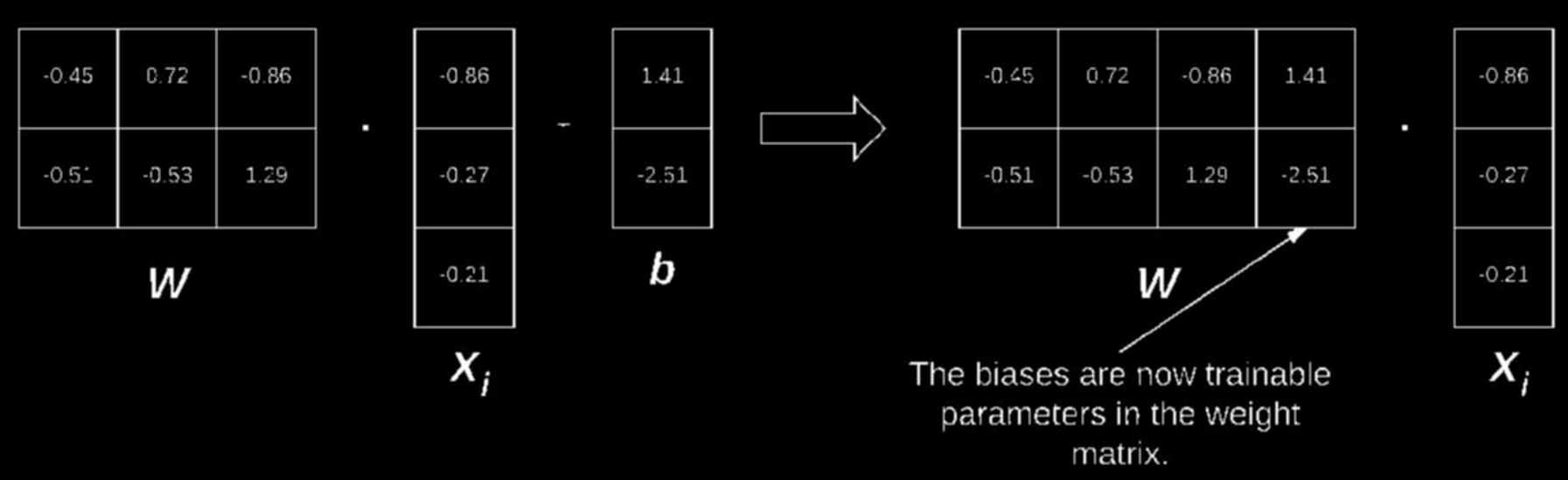

Figure 9.3: **Left:** Normally we treat the weight matrix and bias vector as two separate parameters. **Right:** However, we can actually *embed* the bias vector into the weight matrix (thereby making it a trainable parameter *directly inside* the weight matrix by initializing our weight matrix with an extra column of ones.

To visualize the bias trick, consider Figure 9.3 (*left*) where we *separate* the weight matrix and bias. Up until now, this figure depicts is how we have thought of our scoring function. But instead, we can *combine* the $\boldsymbol{W}$ and $\boldsymbol{b}$ together, provided that we insert a new column into every x_i where every entry is one (Figure 9.3, *right*). Applying the bias trick allows us to learn only a single matrix of weights, hence why we tend to prefer this method for implementation. For all future examples in this book, whenever I mention $\boldsymbol{W}$, assume that the bias vector $\boldsymbol{b}$ is implicitly included in the weight matrix as well.

Pseudocode for Gradient Descent

Below I have included Python-like pseudocode for the standard, vanilla gradient descent algorithm (pseudocode inspired by cs231n slides [83]):

```
1          :
2      Wgradient = evaluate_gradient(loss, data, W)
3      W += -alpha * Wgradient
```

This pseudocode is what *all* variations of gradient descent are built off of. We start off on **Line 1** by looping until some condition is met, typically either:

1. A specified number of epochs has passed (meaning our learning algorithm has "seen" each of the training data points N times).

2. Our loss has become *sufficiently low* or training accuracy *satisfactory high*.
3. Loss has not improved in M subsequent epochs.

Line 2 then calls a function named `evaluate_gradient`. This function requires three parameters:

1. `loss`: A function used to compute the loss over our current parameters `W` and input `data`.
2. `data`: Our training data where each training sample is represented by an image (or feature vector).
3. `W`: Our actual weight matrix that we are optimizing over. Our goal is to apply gradient descent to find a `W` that yields minimal loss.

The `evaluate_gradient` function returns a vector that is K-dimensional, where K is the number of dimensions in our image/feature vector. The `Wgradient` variable is the actual gradient, where we have a gradient entry for each dimension.

We then apply *gradient descent* on **Line 3**. We multiply our `Wgradient` by `alpha` (α), which is our *learning rate*. **The learning rate controls the size of our step.**

In practice, you'll spend *a lot* of time finding an optimal value of α – it is *by far* the most important parameter in your model. If α is too large, you'll spend all of your time bounding around the loss landscape, never actually "descending" to the bottom of the basin (unless your random bouncing takes you there by pure luck). Conversely, if α is too small, then it will take *many* (perhaps prohibitively many) iterations to reach the bottom of the basin. Finding the optimal value of α will cause you many headaches – and you'll send a consider amount of your time trying to find an optimal value for this variable for your model and dataset.

Implementing Basic Gradient Descent in Python

Now that we know the basics of gradient descent, let's implement it in Python and use it to classify some data. Open up a new file, name it `gradient_descent.py`, and insert the following code:

```
1
2       sklearn.model_selection        train_test_split
3       sklearn.metrics        classification_report
4       sklearn.datasets        make_blobs
5         matplotlib.pyplot    plt
6         numpy    np
7         argparse
8
9                        (x):
10
11                / (  + np.exp(-x))
```

Lines 2-7 import our required Python packages. We have seen all of these imports before, with the exception of `make_blobs`, a function used to create "blobs" of normally distributed data points – this is a handy function when testing or implementing our own models from scratch.

We then define the `sigmoid_activation` function on **Line 9**. When plotted this function will resemble an "S"-shaped curve (Figure 9.4). We call it an ***activation function*** because the function will "activate" and fire "ON" (output value > 0.5) or "OFF" (output value $<= 0.5$) based on the inputs `x`.

We can define this relationship via the `predict` method below:

```
13              (X, W):
14
15        preds = sigmoid_activation(X.dot(W))
```

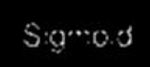

Figure 9.4: The sigmoid activation function. This function is centered at $x = 0.5, y = 0.5$. The function saturates at the tails.

```
16
17
18
19          preds[preds <=     ] =
20          preds[preds >  ] =
21
22
23                  preds
```

Given a set of input data points `X` and weights `W`, we call the `sigmoid_activation` function on them to obtain a set of predictions (**Line 15**). We then threshold the predictions: any prediction with a value $<= 0.5$ is set to `0` while any prediction with a value > 0.5 is set to `1` (**Lines 19 and 20**). The predictions are then returned to the calling function on **Line 23**.

While there are other (better) alternatives to the sigmoid activation function, it makes for an excellent starting point in our discussion of neural networks, deep learning, and gradient-based optimization. I'll be discussing other activation functions in Chapter 10 of the *Starter Bundle* and Chapter 7 of the *Practitioner Bundle*, but for the time being, simply keep in mind that the sigmoid is a non-linear activation function that we can use to threshold our predictions.

Next, let's parse our command line arguments:

```
25
26  ap = argparse.ArgumentParser()
27  ap.add_argument(" e", "  epochs",     =     , default=   ,
28       help="# of epochs")
29  ap.add_argument(" a", "  alpha",     =     , default=    ,
30       help="learning rate")
31  args =     (ap.parse_args())
```

We can provide two (optional) command line arguments to our script:

- `--epochs`: The number of epochs that we'll use when training our classifier using gradient descent.

- `--alpha`: The *learning rate* for the gradient descent. We typically see 0.1, 0.01, and 0.001 as initial learning rate values, but again, this is a hyperparameter you'll need to tune for your own classification problems.

Now that our command line arguments are parsed, let's generate some data to classify:

```
33
34
35 (X, y) = make_blobs(n_samples=    , n_features= , centers= ,
36      cluster_std=   , random_state= )
37 y = y.reshape((y.shape[ ],  ))
38
39
40
41
42 X = np.c_[X, np.ones((X.shape[ ]))]
43
44
45
46 (trainX, testX, trainY, testY) = train_test_split(X, y,
47      test_size=   , random_state=  )
```

On **Line 35** we make a call to `make_blobs` which generates 1,000 data points separated into two classes. These data points are 2D, implying that the "feature vectors" are of length 2. The labels for each of these data points are either 0 or 1. Our goal is to train a classifier that correctly predicts the class label for each data point.

Line 42 applies the "bias trick" (detailed above) that allows us to skip *explicitly* keeping track of our bias vector $\boldsymbol{b}$, by inserting a brand new column of 1s as the last entry in our design matrix `X`. Adding a column containing a constant value across *all* feature vectors allows us to treat our bias as a *trainable parameter* ***within*** the weight matrix $\boldsymbol{W}$ rather than as an entirely separate variable.

Once we have inserted the column of ones, we partition the data into our training and testing splits on **Lines 46 and 47**, using 50% of the data for training and 50% for testing.

Our next code block handles randomly initializing our weight matrix using a uniform distribution such that it has the same number of dimensions as our input features (including the bias):

```
49
50      ("[INFO] training...")
51 W = np.random.randn(X.shape[ ],  )
52 losses = []
```

You might also see both *zero* and *one* weight initialization, but as we'll find out later in this book, good initialization is ***critical*** to training a neural network in a reasonable amount of time, so random initialization along with simple heuristics win out in the vast majority of circumstances [84].

Line 52 initializes a list to keep track of our losses after each epoch. At the end of your Python script, we'll plot the loss (which should ideally decrease over time).

All of our variables are now initialized, so we can move on to the actual training and gradient descent procedure:

```
54
55      epoch    np.arange( , args["epochs"]):
```

```
56
57
58
59        preds = sigmoid_activation(trainX.dot(W))
60
61
62
63
64        error = preds - trainY
65        loss = np.sum(error **  )
66        losses.append(loss)
```

On **Line 55** we start looping over the supplied number of `--epochs`. By default, we'll allow the training procedure to "see" each of the training points a total of 100 times (thus, 100 epochs).

Line 59 takes the dot product between our *entire* training set `trainX` and our weight matrix `W`. The output of this dot product is fed through the sigmoid activation function, yielding our predictions.

Given our predictions, the next step is to determine the "error" of the predictions, or more simply, the difference between our *predictions* and the *true values* (**Line 64**). **Line 65** computes the least squares error over our predictions, a simple loss typically used for binary classification problems. The goal of this training procedure is to minimize our least squares error. We append this `loss` to our `losses` list on **Line 66**, so we can later plot the loss over time.

Now that we have our `error`, we can compute the `gradient` and then use it to update our weight matrix `W`:

```
68
69
70        gradient = trainX.T.dot(error)
71
72
73
74
75
76        W += -args["alpha"] * gradient
77
78
79           epoch ==      (epoch +  ) %   ==  :
80                 ("[INFO] epoch={}, loss={:.7f}".format(   (epoch +  ),
81                 loss))
```

Line 70 handles computing the gradient, which is the dot product between our data points `X` and the `error`.

Line 76 is the most critical step in our algorithm and where the actual gradient descent takes place. Here we update our weight matrix `W` by taking a step in the negative direction of the gradient, thereby allowing us to move towards the bottom of the basin of the loss landscape (hence the term, *gradient descent*). After updating our weight matrix, we check to see if an update should be displayed to our terminal (**Lines 79-81**) and then keep looping until the desired number of epochs has been met – gradient descent is thus an *iterative algorithm*.

Our classifier is now trained. The next step is evaluation:

```
83
84        ("[INFO] evaluating...")
```

```
85 preds = predict(testX, W)
86      (classification_report(testY, preds))
```

To actually make predictions using our weight matrix `W`, we call the `predict` method on `testX` and `W` on **Line 85**. Given the predictions, we display a nicely formatted classification report to our terminal on **Line 86**.

Our last code block handles plotting (1) the *testing data* so we can visualize the dataset we are trying to classify and (2) our loss over time:

```
88
89 plt.style.use("ggplot")
90 plt.figure()
91 plt.title("Data")
92 plt.scatter(testX[:,  ], testX[:,  ], marker="o", c=testY, s=  )
93
94
95 plt.style.use("ggplot")
96 plt.figure()
97 plt.plot(np.arange( , args["epochs"]), losses)
98 plt.title("Training Loss")
99 plt.xlabel("Epoch #")
100 plt.ylabel("Loss")
101 plt.show()
```

Simple Gradient Descent Results

To execute our script, simply issue the following command:

```
python gradient_descent.py
```

As we can see from Figure 9.5 (*left*), our dataset is clearly linear separable (i.e., we can draw a line that separates the two classes of data). Our loss also drops dramatically, starting out very high

and then quickly dropping (*right*). We can see just how quickly the loss drops by investigating the terminal output above. Notice how the loss is initially > 400 but drops to ≈ 1.0 by epoch 50. By the time training terminates by epoch 100, our loss has dropped by an order of magnitude to 0.02.

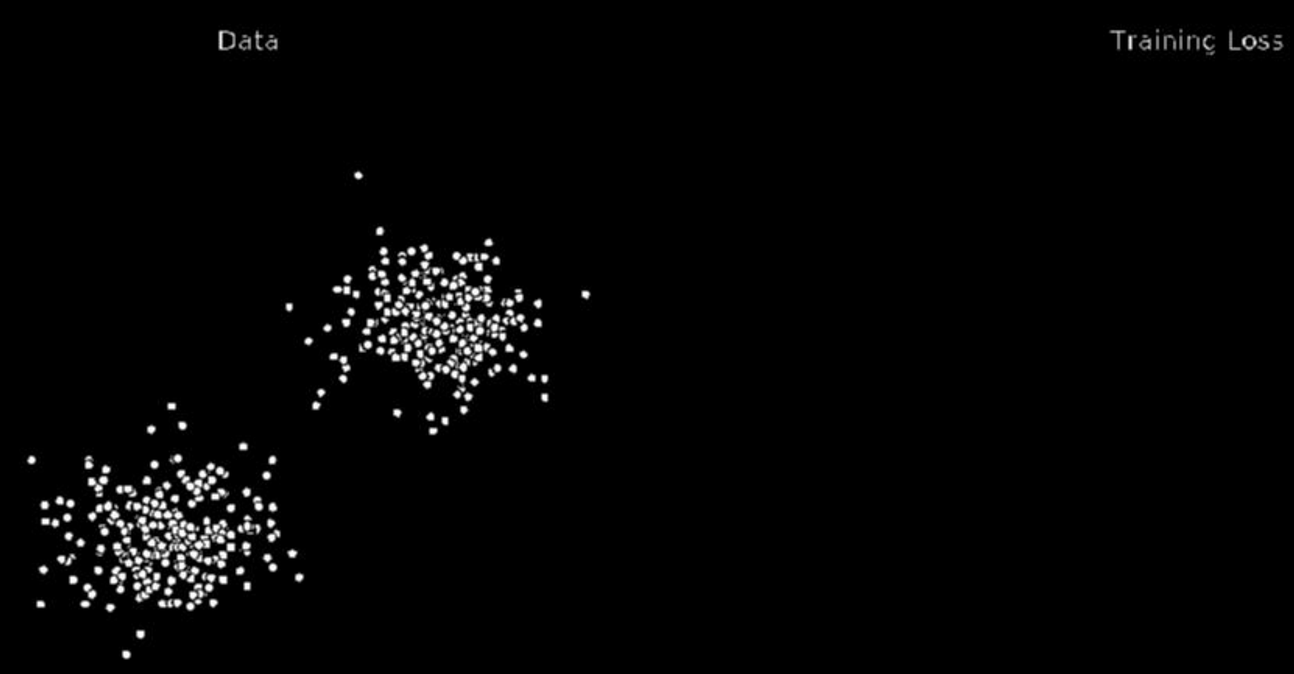

Figure 9.5: **Left:** The input dataset that we are trying to classify into two sets: red and blue. This dataset is clearly linearly separable as we can draw a single line that neatly divides the dataset into two classes. **Right:** Learning a set of parameters to classify our dataset via gradient descent. Loss starts very high but rapidly drops to nearly zero.

This plot validates that our weight matrix is being updated in a manner that allows the classifier to learn from the training data. However, based on the rest of our terminal output, it seems that our classifier misclassified a handful of data points (< 5 of them):

Notice how the zero class is classified correctly 100% of the time, but the one class is classified correctly only 99% of the time. The reason for this discrepancy is because vanilla gradient descent only performs a weight update *once* for every epoch – in this example, we trained our model for 100 epochs, so only 100 updates took place. Depending on the initialization of the weight matrix and the size of the learning rate, it's possible that we might not be able to learn a model that can separate the points (even though they are linearly separable).

In fact, subsequent runs of this script may reveal that *both* classes can be classified correctly 100% of the time – the result is dependent on the initial values `W` takes on. To verify this result yourself, run the `gradient_descent.py` script multiple times.

For simple gradient descent, you are better off training for *more epochs* with a *smaller learning rate* to help overcome this issue. However, as we'll see in the next section, a variant of gradient descent called *Stochastic Gradient Descent* performs a weight update for *every batch* of training data, implying there are *multiple* weight updates per epoch. This approach leads to a faster, more stable convergence.

Stochastic Gradient Descent (SGD)

In the previous section, we discussed gradient descent, a first-order optimization algorithm that can be used to learn a set of classifier weights for parameterized learning. However, this "vanilla" implementation of gradient descent can be prohibitively slow to run on large datasets – in fact, it can even be considered *computational wasteful*.

Instead, we should apply **Stochastic Gradient Descent (SGD)**, a simple modification to the standard gradient descent algorithm that *computes the gradient* and *updates the weight matrix* $\boldsymbol{W}$ on **small batches of training data**, rather than the entire training set. While this modification leads to "more noisy" updates, it also allows us to take *more steps along the gradient* (one step per each batch versus one step per epoch), ultimately leading to faster convergence and no negative affects to loss and classification accuracy.

SGD is arguably ***the most important algorithm*** when it comes to training deep neural networks. Even though the original incarnation of SGD was introduced over 57 years ago [85], it is *still* the engine that enables us to train large networks to learn patterns from data points. Above all other algorithms covered in this book, take the time to understand SGD.

Mini-batch SGD

Reviewing the vanilla gradient descent algorithm, it should be (somewhat) obvious that the method will run *very slowly* on large datasets. The reason for this slowness is because each iteration of gradient descent requires us to compute a prediction for each training point in our training data *before* we are allowed to update our weight matrix. For image datasets such as ImageNet were we have over *1.2 million* training images, this computation can take a long time.

It also turns out that computing predictions for *every* training point before taking a step along our weight matrix is computationally wasteful and does little to help our model coverage.

Instead, what we should do is *batch* our updates. We can update the pseudocode to transform vanilla gradient descent to become SGD by adding an extra function call:

```
        :
    batch = next_training_batch(data,    )
    Wgradient = evaluate_gradient(loss, batch, W)
    W += -alpha * Wgradient
```

The only difference between vanilla gradient descent and SGD is the addition of the `next_training_batch` function. Instead of computing our gradient over the *entire* `data` set, we instead sample our data, yielding a `batch`. We evaluate the gradient on the `batch`, and update our weight matrix `W`. From an implementation perspective, we also try to randomize our training samples *before* applying SGD since the algorithm is sensitive to batches.

After looking at the pseudocode for SGD, you'll immediately notice an introduction of a new parameter: ***the batch size***. In a "purist" implementation of SGD, your mini-batch size would be 1, implying that we would randomly sample *one* data point from the training set, compute the gradient, and update our parameters. However, we often use mini-batches that are > 1. Typical batch sizes include 32, 64, 128, and 256.

So, why bother using batch sizes > 1? To start, batch sizes > 1 help reduce variance in the parameter update ( ), leading to a more stable convergence. Secondly, powers of two are often desirable for batch sizes as they allow internal linear algebra optimization libraries to be more efficient.

In general, the mini-batch size is not a hyperparameter you should worry too much about [57]. If you're using a GPU to train your neural network, you determine how many training examples will fit into your GPU and then use the nearest power of two as the batch size such that the batch

will fit on the GPU. For CPU training, you typically use one of the batch sizes listed above to ensure you reap the benefits of linear algebra optimization libraries.

Implementing Mini-batch SGD

Let's go ahead and implement SGD and see how it differs from standard vanilla gradient descent. Open up a new file, name it `sgd.py`, and insert the following code:

```
1
2       sklearn.model_selection        train_test_split
3       sklearn.metrics        classification_report
4       sklearn.datasets        make_blobs
5         matplotlib.pyplot    plt
6         numpy    np
7         argparse
8
9                        (x):
10
11                / (  + np.exp(-x))
```

Lines 2-7* import our required Python packages, exactly the same as the `gradient_descent.py` example earlier in this chapter. **Lines 9-11** define the `sigmoid_activation` function, which is also identical to the previous version of gradient descent.

In fact, the `predict` method doesn't change either:

```
13            (X, W):
14
15      preds = sigmoid_activation(X.dot(W))
16
17
18
19      preds[preds <=    ] =
20      preds[preds >  ] =
21
22
23            preds
```

However, what *does* change is the addition of the `next_batch` function:

```
25              (X, y, batchSize):
26
27
28          i    np.arange( , X.shape[ ], batchSize):
29                (X[i:i + batchSize], y[i:i + batchSize])
```

The `next_batch` method requires three parameters:

1. `X`: Our training dataset of feature vectors/raw image pixel intensities.
2. `y`: The class labels associated with each of the training data points.
3. `batchSize`: The size of each mini-batch that will be returned.

Lines 28 and 29 then loop over the training examples, yielding subsets of both `X` and `y` as mini-batches.

Next, we can parse our command line arguments:

```

ap = argparse.ArgumentParser()
ap.add_argument(" e", "  epochs",     =     , default=   ,
     help="# of epochs")
ap.add_argument(" a", "  alpha",     =     , default=    ,
     help="learning rate")
ap.add_argument(" b", "  batch size",     =   , default=  ,
     help="size of SGD mini batches")
args =     (ap.parse_args())
```

We have already reviewed both the `--epochs` (number of epochs) and `--alpha` (learning rate) switch from the vanilla gradient descent example – but also notice we are introducing a third switch: `--batch-size`, which as the name indicates is the size of each of our mini-batches. We'll default this value to be 32 data points per mini-batch.

Our next code block handles generating our 2-class classification problem with 1,000 data points, adding the bias column, and then performing the training and testing split:

```


(X, y) = make_blobs(n_samples=    , n_features= , centers= ,
     cluster_std=   , random_state= )
y = y.reshape((y.shape[ ],  ))



X = np.c_[X, np.ones((X.shape[ ]))]



(trainX, testX, trainY, testY) = train_test_split(X, y,
     test_size=   , random_state=  )
```

We'll then initialize our weight matrix and losses just like in the previous example:

```

     ("[INFO] training...")
W = np.random.randn(X.shape[ ],  )
losses = []
```

The *real* change comes next where we loop over the desired number of epochs, sampling mini-batches along the way:

```

    epoch    np.arange( , args["epochs"]):

     epochLoss = []


         (batchX, batchY)    next_batch(X, y, args["batch_size"]):

```

17.1 What Are Underfitting and Overfitting? 251
17.1.1 Effects of Learning Rates 253
17.1.2 Pay Attention to Your Training Curves 254
17.1.3 What if Validation Loss Is Lower than Training Loss? 254
17.2 Monitoring the Training Process 255
17.2.1 Creating a Training Monitor 255
17.2.2 Babysitting Training 257
17.3 Summary 260

18.1 Checkpointing Neural Network Model Improvements 263
18.2 Checkpointing Best Neural Network Only 267
18.3 Summary 269

19.1 The Importance of Architecture Visualization 271
19.1.1 Installing graphviz and pydot 272
19.1.2 Visualizing Keras Networks 272
19.2 Summary 275

20.1 State-of-the-art CNNs in Keras 277
20.1.1 VGG16 and VGG19 278
20.1.2 ResNet 279
20.1.3 Inception V3 280
20.1.4 Xception 280
20.1.5 Can We Go Smaller? 280
20.2 Classifying Images with Pre-trained ImageNet CNNs 281
20.2.1 Classification Results 284
20.3 Summary 286

21.1 Breaking Captchas with a CNN 288
21.1.1 A Note on Responsible Disclosure 288
21.1.2 The Captcha Breaker Directory Structure 290
21.1.3 Automatically Downloading Example Images 291
21.1.4 Annotating and Creating Our Dataset 292
21.1.5 Preprocessing the Digits 297
21.1.6 Training the Captcha Breaker 299
21.1.7 Testing the Captcha Breaker 303
21.2 Summary 305

22.1 The SMILES Dataset 307

```
70
71
72          preds = sigmoid_activation(batchX.dot(W))
73
74
75
76
77          error = preds - batchY
78          epochLoss.append(np.sum(error **  ))
```

On **Line 63** we start looping over the supplied number of `--epochs`. We then loop over our training data in batches on **Line 68**. For each batch, we compute the dot product between the batch and `W`, then pass the result through the sigmoid activation function to obtain our predictions. We compute the least square error for the batch on **Line 77** and use this value to update our `epochLoss` on **Line 78**.

Now that we have the `error`, we can compute the gradient descent update, which is the dot product between the current batch data points and the error on the batch:

```
80
81
82          gradient = batchX.T.dot(error)
83
84
85
86
87
88          W += -args["alpha"] * gradient
```

Line 88 handles updating our weight matrix based on the gradient, scaled by our learning rate `--alpha`. Notice how the weight update stage takes place *inside* the batch loop – this implies there are *multiple weight updates per epoch*.

We can then update our loss history by taking the average across all batches in the epoch and then displaying an update to our terminal if necessary:

```
90
91
92      loss = np.average(epochLoss)
93      losses.append(loss)
94
95
96         epoch ==      (epoch +  ) %   ==  :
97                ("[INFO] epoch={}, loss={:.7f}".format(   (epoch +  ),
98                loss))
```

Evaluating our classifier is done in the same way as in vanilla gradient descent – simply call `predict` on the `testX` data using our learned `W` weight matrix:

```
100
101     ("[INFO] evaluating...")
102 preds = predict(testX, W)
103     (classification_report(testY, preds))
```

We'll end our script by plotting the testing classification data and along with the loss per epoch:

```

plt.style.use("ggplot")
plt.figure()
plt.title("Data")
plt.scatter(testX[:,  ], testX[:,  ], marker="o", c=testY, s=  )


plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange( , args["epochs"]), losses)
plt.title("Training Loss")
plt.xlabel("Epoch #")
plt.ylabel("Loss")
plt.show()
```

SGD Results

To visualize the results from our implementation, just execute the following command:

```
python sgd.py
```

We'll be using the same "blob" dataset as in Figure 9.5 (*left*) above for classification so we can compare our SGD results to vanilla gradient descent. Furthermore, SGD example uses the same

learning rate (0.1) and the same number of epochs (100) as vanilla gradient descent. However, notice how much *smoother* our loss curve is in Figure 9.6.

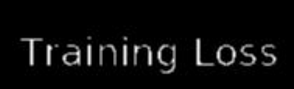

Figure 9.6: Applying Stochastic Gradient Descent to our dataset of red and blue data points. Using SGD, our learning curve is *much* smoother. Furthermore, we are able to obtain an order of magnitude lower loss by the end of the 100th epoch (as compared to standard, vanilla gradient descent).

Investigating the actual loss values at the end of the 100th epoch, you'll notice that loss obtained by SGD is ***an order of magnitude lower*** than vanilla gradient descent (0.003 vs 0.021, respectively). This difference is due to the multiple weight updates per epoch, giving our model more chances to learn from the updates made to the weight matrix. This effect is even more pronounced on large datasets, such as ImageNet where we have millions of training examples and small, incremental updates in our parameters can lead to a low loss (but not necessarily optimal) solution.

Extensions to SGD

There are two primary extensions that you'll encounter to SGD in practice. The first is momentum [86], a method used to accelerate SGD, enabling it to learn faster by focusing on dimensions whose gradient point in the same direction. The second method is Nesterov acceleration [87], an extension to standard momentum.

Momentum

Consider your favorite childhood playground where you spent days rolling down a hill, covering yourself in grass and dirt (much to your mother's chagrin). As you travel down the hill, you build up more and more momentum, which in turn carries you faster down the hill.

Momentum applied to SGD has the same effect – our goal is to build upon the standard weight update to include a momentum term, thereby allowing our model to obtain lower loss (and higher

accuracy) in less epochs. The momentum term should, therefore, *increase* the strength of updates for dimensions who gradients point in the same direction and then *decrease* the strength of updates for dimensions who gradients switch directions [86, 88].

Our previous weight update rule simply included the scaling the gradient by our learning rate:

$$W = W - \alpha \nabla_{W} f(W) \tag{9.4}$$

We now introduce the momentum term V, scaled by γ:

$$V = \gamma V - \alpha \nabla_{W} f(W) \qquad W = W + V \tag{9.5}$$

The momentum term γ is commonly set to 0.9; although another common practice is to set γ to 0.5 until learning stabilizes and then increase it to 0.9 – it is extremely rare to see momentum < 0.5. For a more detailed review of momentum, please refer to Sutton [89] and Qian [86].

Nesterov's Acceleration

Let's suppose that you are back on your childhood playground, rolling down the hill. You've built up momentum and are moving quite fast – but there's a problem. At the bottom of the hill is the brick wall of your school, one that you would like to avoid hitting at full speed.

The same thought can be applied to SGD. If we build up too much momentum, we may overshoot a local minimum and keep on rolling. Therefore, it would be advantageous to have a smarter roll, one that knows when to slow down, which is where Nesterov accelerated gradient [87] comes in.

Nesterov acceleration can be conceptualized as a corrective update to the momentum which lets us obtain an approximate idea of where our parameters will be after the update. Looking at Hinton's *Overview of mini-batch gradient descent* slides [90], we can see a nice visualization of Nesterov acceleration (Figure 9.7).

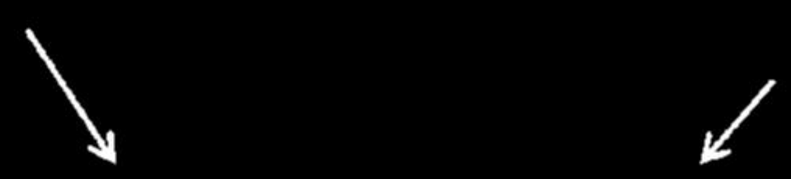

Figure 9.7: A graphical depecition of Nesterov acceleration. First, we make a big jump in the direction of the previous gradient, then measure the gradient where we ended up and make the correction.

Using standard momentum, we compute the gradient (small blue vector) and then take a big jump in the in the direction of the gradient (large blue vector). Under Nesterov acceleration we would first make a big jump in the direction of our *previous* gradient (brown vector), measure the gradient, and then make a *correction* (red vector) – the green vector is the final corrected update by Nesterov acceleration (paraphrased from Ruder [88]).

A thorough theoretical and mathematical treatment of Nesterov acceleration are outside the scope of this book. For those interested in studying Nesterov acceleration in more detail, please refer to Ruder [88], Bengio [91], and Sutskever [92].

Anecdotal Recommendations

Momentum is an important term that can increase the convergence of our model; we tend not to worry with this hyperparameter as much, as compared to our learning rate and regularization penalty (discussed in the next section), which are *by far* the most important knobs to tweak.

My personal rule of thumb is that whenever using SGD, also apply momentum. In most cases, you can set it (and leave it) and 0.9 although Karpathy [93] suggests starting at 0.5 and increasing it to larger values as your epochs increase.

As for Nesterov acceleration, I tend to use it on smaller datasets, but for larger datasets (such as ImageNet), I almost always avoid it. While Nesterov acceleration has sound theoretical guarantees, all major publications trained on ImageNet (e.x., AlexNet [94], VGGNet [95], ResNet [96], Inception [97], etc.) use SGD with momentum – *not a single paper from this seminal group utilizes Nesterov acceleration.*

My personal experience has lead me to find that when training deep networks on large datasets, SGD is easier to work with when using momentum and *leaving out* Nesterov acceleration. Smaller datasets, on the other hand, tend to enjoy the benefits of Nesterov acceleration. However, keep in mind that this is my anecdotal opinion and that your mileage may vary.

Regularization

> *"Many strategies used in machine learning are explicitly designed to reduce the test error, possibly at the expense of increased training error. These strategies are collectively known as regularization."* – Goodfellow et al. [10]

In earlier sections of this chapter, we discussed two important loss functions: Multi-class SVM loss and cross-entropy loss. We then discussed gradient descent and how a network can actually learn by updating the weight parameters of a model. While our loss function allows us to determine how well (or poorly) our set of parameters are performing on a given classification task, the loss function itself does not take into account how the weight matrix "looks".

What do I mean by "looks"? Well, keep in mind that we are working in a real-valued space, thus there are an *infinite set* of parameters that will obtain reasonable classification accuracy on our dataset (for some definition of "reasonable").

How do we go about choosing a set of parameters that help ensure our model generalizes well? Or, at the very least, lessen the effects of overfitting. ***The answer is regularization.*** Second only to your learning rate, regularization is the most important parameter of your model that can you tune.

There are various types of regularization techniques, such as L1 regularization, L2 regularization (commonly called "weight decay"), and Elastic Net [98], that are used by updating the loss function itself, adding an additional parameter to constrain the capacity of the model.

We also have types of regularization that can be *explicitly* added to the network architecture – dropout is the quintessential example of such regularization. We then have *implicit* forms of regularization that are applied during the training process. Examples of implicit regularization include data augmentation and early stopping. Inside this section, we'll mainly be focusing on the parameterized regularization obtained by modifying our loss and update functions.

In Chapter 11 of the *Starter Bundle*, we'll review dropout and then in Chapter 17 we'll discuss overfitting in more depth, as well as how we can use early stopping as a regularizer. Inside the *Practitioner Bundle*, you'll find examples of data augmentation used as regularization.

What Is Regularization and Why Do We Need It?

Regularization helps us control our model capacity, ensuring that our models are better at making (correct) classifications on data points that they were *not* trained on, which we call ***the***

ability to generalize. If we don't apply regularization, our classifiers can easily become too complex and *overfit* to our training data, in which case we lose the ability to generalize to our testing data (and data points *outside* the testing set as well, such as new images in the wild).

However, too much regularization can be a bad thing. We can run the risk of *underfitting*, in which case our model performs poorly on the training data and is not able to model the relationship between the input data and output class labels (because we limited model capacity too much). For example, consider the following plot of points, along with various functions that fit to these points (Figure 9.8).

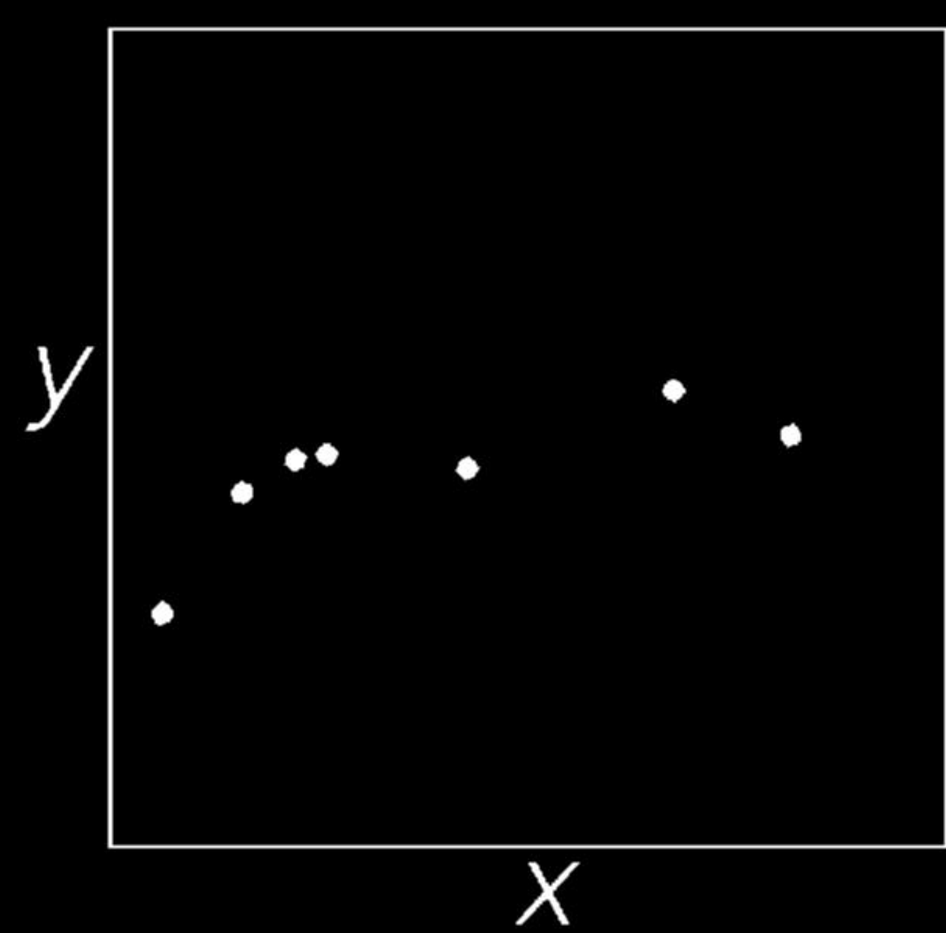

Figure 9.8: An example of underfitting (orange line), overfitting (blue line), and generalizing (green line). Our goal when building deep learning classifiers is to obtain these types of "green functions" that fit our training data nicely, but avoid overfitting. Regularization can help us obtain this type of desired fit.

The orange line is an example of *underfitting* – we are not capturing the relationship between the points. On the other hand, the blue line is an example of *overfitting* – we have too many parameters in our model, and while it hits all points in the dataset, it also wildly varies between the points. It is not a smooth, simple fit that we would prefer. We then have the green function which also hits all points in our dataset, but does so in a much more predictable, simple manner.

The goal of regularization is to obtain these types of "green functions" that fit our training data nicely, but avoid overfitting to our training data (blue) or failing to model the underlying relationship (yellow). We discuss how to monitor training and spot both underfitting and overfitting in Chapter 17; however, for the time being, simply understand that regularization is a critical aspect of machine learning and we use regularization to control model generalization. To understand regularization and the impact it has on our loss function and weight update rule, let's proceed to the next section.

Updating Our Loss and Weight Update To Include Regularization

Let's start with our cross-entropy loss function (Section 8.2.3):

$$L_i = -log(e^{s_{y_i}} / \sum_j e^{s_j}) \tag{9.6}$$

The loss over the entire training set can be written as:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i \tag{9.7}$$

Now, let's say that we have obtained a weight matrix $\boldsymbol{W}$ such that *every data point* in our training set is classified correctly, which means that our loss $L = 0$ for all L_i.

Awesome, we're getting 100% accuracy – but let me ask you a question about this weight matrix – ***is it unique?*** **Or, in other words, are there *better* choices of $\boldsymbol{W}$ that will improve our model's ability to generalize and reduce overfitting?**

If there is such a $\boldsymbol{W}$, how do we know? And how can we incorporate this type of penalty into our loss function? The answer is to define a **regularization penalty**, a function that operates on our weight matrix. The regularization penalty is commonly written as a function, $R(\boldsymbol{W})$. Equation 9.8 below shows the most common regularization penalty, L2 regularization (also called **weight decay**):

$$R(W) = \sum_i \sum_j W_{i,j}^2 \tag{9.8}$$

What is the function doing exactly? In terms of Python code, it's simply taking the sum of squares over an array:

```
penalty =

    i    np.arange( , W.shape[ ]):
         j    np.arange( , W.shape[ ]):
          penalty += (W[i][j] **  )
```

What we are doing here is looping over all entries in the matrix and taking the sum of squares. The sum of squares in the L2 regularization penalty discourages large weights in our matrix $\boldsymbol{W}$, preferring smaller ones. Why might we want to discourage large weight values? In short, by penalizing large weights, we can improve the ability to generalize, and thereby reduce overfitting.

Think of it this way – the larger a weight value is, the more influence it has on the output prediction. Dimensions with larger weight values can almost singlehandedly control the output prediction of the classifier (provided the weight value is large enough, of course) which will almost certainly lead to overfitting.

To mitigate affect various dimensions have on our output classifications, we apply regularization, thereby seeking $\boldsymbol{W}$ values that take into account *all* of the dimensions rather than the few with large values. In practice you may find that regularization hurts your training accuracy slightly, but actually *increases your testing accuracy*.

Again, our loss function has the same basic form, only now we add in regularization:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i - \lambda R(W) \tag{9.9}$$

The first term we have seen before – it is the average loss over all samples in our training set.

The second term is new – *this is our regularization penalty*. The λ variable is a hyperparameter that controls the *amount* or *strength* of the regularization we are applying. In practice, both the learning rate α and the regularization term λ are the hyperparameters that you'll spending the most time tuning.

Expanding cross-entropy loss to include L2 regularization yields the following equation:

$$L = \frac{1}{N}\sum_{i=1}^{N}|-log(e^{s_{y_i}}/\sum_j e^{s_j})| + \lambda\sum_i\sum_j W_{i,j}^2 \tag{9.10}$$

We can also expand Multi-class SVM loss as well:

$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq y_i}|max(0, s_j - s_{y_i} + 1)| + \lambda\sum_i\sum_j W_{i,j}^2 \tag{9.11}$$

Now, let's take a look at our standard weight update rule:

$$W = W - \alpha\nabla_W f(W) \tag{9.12}$$

This method updates our weights based on the gradient multiple by a learning rate α. Taking into account regularization, the weight update rule becomes:

$$W = W - \alpha\nabla_W f(W) + \lambda R(W) \tag{9.13}$$

Here we are adding a negative linear term to our gradients (i.e., gradient descent), penalizing large weights, with the end goal of making it easier for our model to generalize.

Types of Regularization Techniques

In general, you'll see three common types of regularization there are applied directly to the loss function. The first, we reviewed earlier, L2 regularization (aka "weight decay"):

$$R(W) = \sum_i\sum_j W_{i,j}^2 \tag{9.14}$$

We also have L1 regularization which takes the absolute value rather than the square:

$$R(W) = \sum_i\sum_j |W_{i,j}| \tag{9.15}$$

Elastic Net [98] regularization seeks to combine both L1 and L2 regularization:

$$R(W) = \sum_i\sum_j \beta W_{i,j}^2 + |W_{i,j}| \tag{9.16}$$

Other types of regularization methods exist such as directly modifying the architecture of a network along with how the network is actually trained – we will review these methods in later chapters.

In terms of *which* regularization method you should be using (including none at all), you should treat this choice as a hyperparameter you need to optimize over and perform experiments to determine *if* regularization should be applied, and if so *which method* of regularization, and what the proper value of λ is. For more details on regularization, refer to Chapter 7 of Goodfellow et al. [10], the "Regularization" section from the DeepLearing.net tutorial [99], and the notes from Karpathy's cs231n Neural Networks II lecture [100].

Regularization Applied to Image Classification

To demonstrate regularization in action, let's write some Python code to apply it to our "Animals" dataset. Open up a new file, name it `regularization.py`, and insert the following code:

```
1
2       sklearn.linear_model        SGDClassifier
3       sklearn.preprocessing        LabelEncoder
4       sklearn.model_selection        train_test_split
5       pyimagesearch.preprocessing        SimplePreprocessor
6       pyimagesearch.datasets        SimpleDatasetLoader
7       imutils        paths
8         argparse
```

Lines 2-8 import our required Python packages. We've seen all of these imports before, except the scikit-learn `SGDClassifier`. As the name of this class suggests, this implementation encapsulates all the concepts we have reviewed in this chapter, including:

- Loss functions
- Number of epochs
- Learning rate
- Regularization terms

Thus making it the perfect example to demonstrate all these concepts in action.

Next, we can parse our command line arguments and grab the list of images from disk:

```
10
11  ap = argparse.ArgumentParser()
12  ap.add_argument(" d", "  dataset", required=    ,
13       help="path to input dataset")
14  args =     (ap.parse_args())
15
16
17       ("[INFO] loading images...")
18  imagePaths =     (paths.list_images(args["dataset"]))
```

Given the image paths, we'll resize them to 32×32 pixels, load them from disk into memory, and then flatten them into a 3,072-dim array:

```
20
21
22  sp = SimplePreprocessor(  ,   )
23  sdl = SimpleDatasetLoader(preprocessors=[sp])
24  (data, labels) = sdl.load(imagePaths, verbose=   )
25  data = data.reshape((data.shape[ ],     ))
```

We'll also encode the labels as integers and perform a training testing split, using 75% of the data for training and the remaining 25% for testing:

```
27
28  le = LabelEncoder()
29  labels = le.fit_transform(labels)
30
31
32
33  (trainX, testX, trainY, testY) = train_test_split(data, labels,
34      test_size=    , random_state= )
```

Let's apply a few different types of regularization when training our `SGDClassifier`:

```
36
37      r    (    , "l1", "l2"):
38
39
40          ("[INFO] training model with '{}' penalty".format(r))
41      model = SGDClassifier(loss="log", penalty=r, max_iter=  ,
42          learning_rate="constant", eta0=    , random_state=  )
43      model.fit(trainX, trainY)
44
45
46      acc = model.score(testX, testY)
47          ("[INFO] '{}' penalty accuracy: {:.2f}%".format(r,
48          acc *    ))
```

Line 37 loops over our regularizers, including no regularization. We then initialize and train the `SGDClassifier` on **Lines 41-43**.

We'll be using cross-entropy loss, with regularization penalty of `r` and a default λ of 0.0001. We'll use SGD to train the model for 10 epochs with a learning rate of $\alpha = 0.01$. We then evaluate the classifier and display the accuracy results to our screen on **Lines 46-48**.

To see our SGD model trained with various regularization types, just execute the following command:

```
python regularization.py --dataset ../datasets/animals
```

We can see with *no regularization* we obtain an accuracy of **50.40%**. Using L1 regularization our accuracy increases to **52.53%**. L2 regularization obtains the highest accuracy of **55.07%**.

Using different `random_state` values for `train_test_split` will yield different results. The dataset here is too small and the classifier too simplistic to see the full impact of

22.2 Training the Smile CNN 308
22.3 Running the Smile CNN in Real-time 313
22.4 Summary 316

23.1 So, What's Next? 319

regularization, so consider this a "worked example". As we continue to work through this book you'll see more advanced uses of regularization that will have dramatic impacts on your accuracy.

Realistically, this example is too small to show all the advantages of applying regularization – for that, we'll have to wait until we start training Convolutional Neural Networks. However, in the meantime simply appreciate that regularization can provide a boost in our testing accuracy and reduce overfitting, *provided we can tune the hyperparameters right*.

Summary

In this chapter, we popped the hood on deep learning and took a deep dive into the engine that powers modern day neural networks – *gradient descent*. We investigated two types of gradient descent:

1. The standard vanilla flavor.
2. The stochastic version that is more commonly used.

Vanilla gradient descent performs only *one* weight update per epoch, making it very slow (if not impossible) to converge on large datasets. The stochastic version instead applies *multiple* weight updates per epoch by computing the gradient on small mini-batches. By using SGD we can dramatically reduce the time it takes to train a model while also enjoying lower loss and higher accuracy. Typical batch sizes include 32, 64, 128 and 256.

Gradient descent algorithms are controlled via a ***learning rate***: this is by far the most important parameter to tune correctly when training your own models.

If your learning rate is too large, you'll simply bounce around the loss landscape and not actually "learn" any patterns from your data. On the other hand, if your learning rate is too small, it will take a prohibitive number of iterations to reach even a reasonable loss. To get it just right, you'll want to spend the majority of your time tuning the learning rate.

We then discussed ***regularization***, which is defined as *"any method that increases testing accuracy perhaps at the expense of training accuracy"*. Regularization encompasses a broad range of techniques. We specifically focused on regularization methods that are applied to our loss functions and weight update rules, including L1 regularization, L2 regularization, and Elastic Net.

In terms of deep learning and neural networks, you'll commonly see L2 regularization used for image classification – the trick is tuning the λ parameter to include just the right amount of regularization.

At this point, we have a sound foundation of machine learning, but we have yet to investigate neural networks or train a custom neural network from scratch. That will all change in our next chapter where we discuss neural networks, the backpropagation algorithm, and how to train your own neural networks on custom datasets.

10. Neural Network Fundamentals

In this chapter, we'll study the fundamentals of neural networks in depth. We'll start with a discussion of artificial neural networks and how they are inspired by the real-life biological neural networks in our own bodies. From there, we'll review the classic Perceptron algorithm and the role it has played in neural network history.

Building on the Perceptron, we'll also study the *backpropagation algorithm*, the cornerstone of modern neural learning – without backpropagation, we would be unable to efficiently train our networks. We'll also implement backpropagation with Python from scratch, ensuring we understand this important algorithm.

Of course, modern neural network libraries such as Keras already have (highly optimized) backpropagation algorithms built-in. Implementing backpropagation by hand each time we wished to train a neural network would be like coding a linked list or hash table data structure from scratch each time we worked on a general purpose programming problem – not only is it unrealistic, but it's also a waste of our time and resources. In order to streamline the process, I'll demonstrate how to create standard feedforward neural networks using the Keras library.

Finally, we'll round out this chapter with a discussion of the four ingredients you'll need when building *any* neural network.

Neural Network Basics

Before we can work with Convolutional Neural Networks, we first need to understand the basics of neural networks. In this section we'll review:

- Artificial Neural Networks and their relation to biology.
- The seminal Perceptron algorithm.
- The backpropagation algorithm and how it can be used to train *multi-layer* neural networks efficiently.
- How to train neural networks using the Keras library.

By the time you finish this chapter, you'll have a strong understanding of neural networks and be able to move on to the more advanced Convolutional Neural Networks.

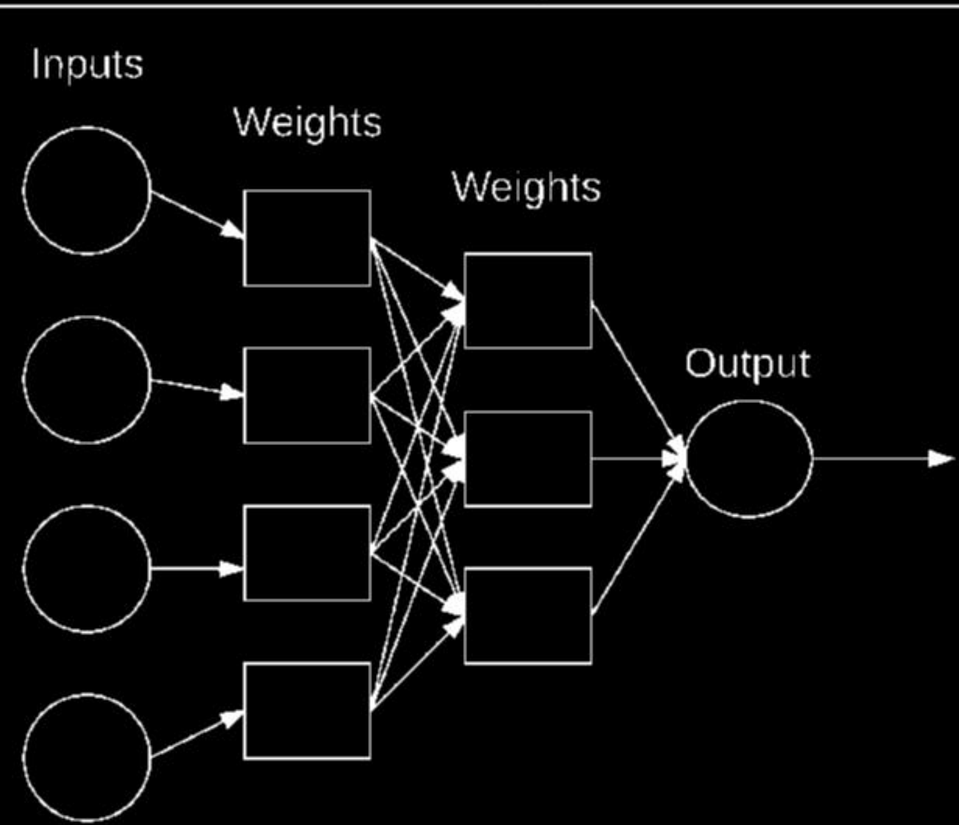

Figure 10.1: A simple neural network architecture. Inputs are presented to the network. Each connection carries a signal through the two hidden layers in the network. A final function computes the output class label.

Introduction to Neural Networks

Neural networks are the building blocks of deep learning systems. In order to be successful at deep learning, we need to start by reviewing the basics of neural networks, including *architecture*, *node types*, and *algorithms for "teaching"* our networks.

In this section we'll start off with a high-level overview of neural networks and the motivation behind them, including their relation to biology in the human mind. From there we'll discuss the most common type of architecture, **feedforward neural networks**. We'll also briefly discuss the concept of *neural learning* and how it will later relate to the algorithms we use to train neural networks.

What are Neural Networks?

Many tasks that involve intelligence, pattern recognition, and object detection are *extremely difficult to automate*, yet *seem to be performed easily and naturally* by animals and young children. For example, how does your family dog recognize you, the owner, versus a complete and total stranger? How does a small child learn to recognize the difference between a *school bus* and a *transit bus*? And how do our own brains subconsciously perform complex pattern recognition tasks each and every day without us even noticing?

The answer lies within our own bodies. Each of us contains a real-life biological neural networks that is connected to our nervous systems – this network is made up of a large number of interconnected *neurons* (nerve cells).

The word *"neural"* is the adjective form of *"neuron"*, and *"network"* denotes a graph-like structure; therefore, an *"Artificial Neural Network"* is a computation system that attempts to mimic (or at least, is inspired by) the neural connections in our nervous system. Artificial neural networks are also referred to as *"neural networks"* or *"artificial neural systems"*. It is common to abbreviate Artificial Neural Network and refer to them as *"ANN"* or simply *"NN"* – I will be using both of the abbreviations throughout the rest of the book.

For a system to be considered an NN, it must contain a labeled, directed graph structure where each node in the graph performs some *simple computation*. From graph theory, we know that a directed graph consists of a set of nodes (i.e., vertices) and a set of connections (i.e., edges) that link together pairs of nodes. In Figure 10.1 we can see an example of such an NN graph.

Each node performs a simple computation. Each connection then carries a *signal* (i.e., the output of the computation) from one node to another, labeled by a *weight* indicating the extent to which the signal is amplified or diminished. Some connections have large, *positive weights* that amplify the signal, indicating that the signal is very important when making a classification. Others have *negative weights*, diminishing the strength of the signal, thus specifying that the output of the node is less important in the final classification. We call such a system an *Artificial Neural Network* if it consists of a graph structure (like in Figure 10.1) with connection weights that are *modifiable* using a learning algorithm.

Relation to Biology

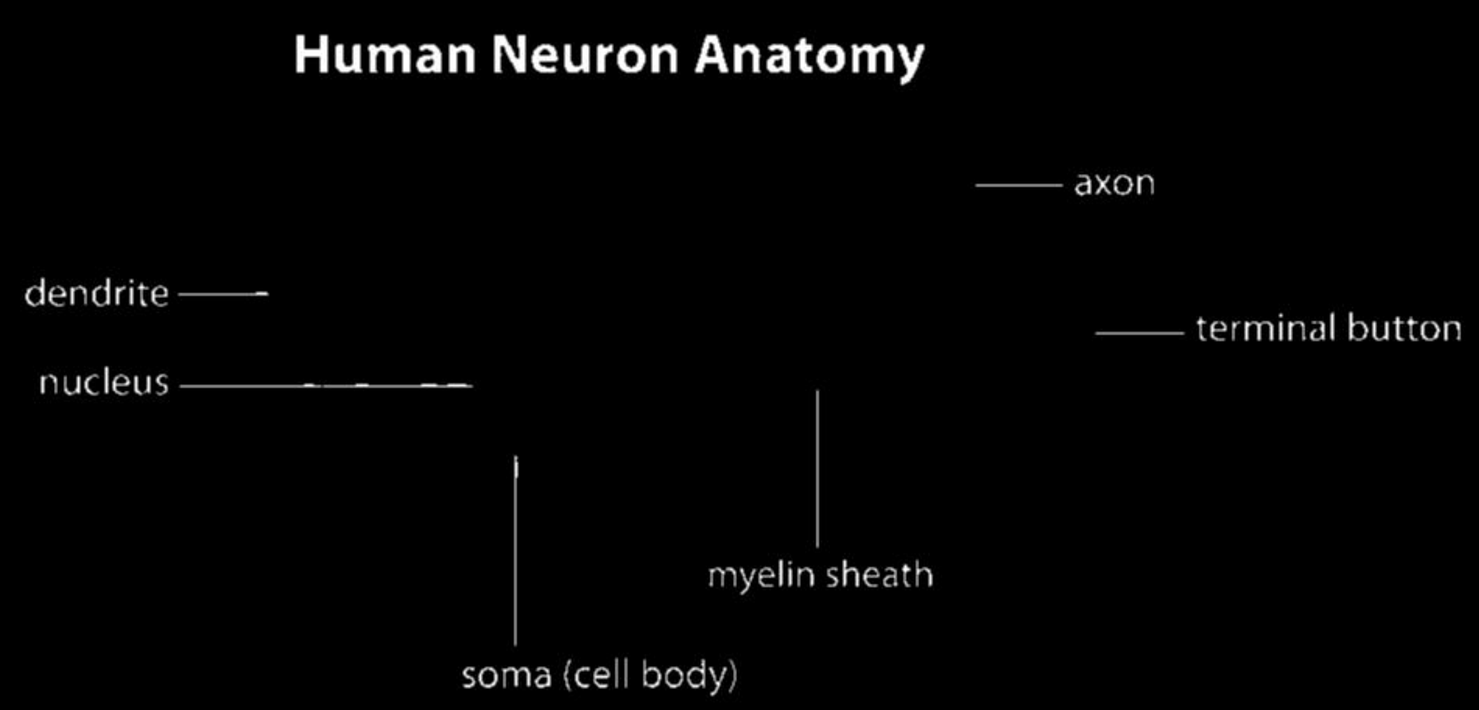

Figure 10.2: The structure of a biological neuron. Neurons are connected to other neurons through their dendrites and enurons.

Our brains are composed of approximately 10 billion neurons, each connected to about 10,000 other neurons. The cell body of the neuron is called the *soma*, where the inputs (*dendrites*) and outputs (*axons*) connect soma to other soma (Figure 10.2).

Each neuron receives electrochemical inputs from other neurons at their dendrites. If these electrical inputs are sufficiently powerful to activate the neuron, then the activated neuron transmits the signal along its axon, passing it along to the dendrites of other neurons. These attached neurons may also fire, thus continuing the process of passing the message along.

The key takeaway here is that a neuron firing is a **binary operation – the neuron either *fires* or it *doesn't fire***. There are no different "grades" of firing. Simply put, a neuron will only fire if the total signal received at the soma exceeds a given threshold.

However, keep in mind that ANNs are simply *inspired* by what we know about the brain and how it works. The goal of deep learning is *not* to mimic how our brains function, but rather take the pieces that we *understand* and allow us to draw similar parallels in our own work. At the end of the day we do not know enough about neuroscience and the deeper functions of the brain to be able to correctly model how the brain works – instead, we take our *inspirations* and move on from there.

Artificial Models

Let's start by taking a look at a basic NN that performs a simple weighted summation of the inputs in Figure 10.3. The values x_1, x_2, and, x_3 are the **inputs** to our NN and typically correspond to a *single row* (i.e., data point) from our design matrix. The constant value 1 is our bias that is assumed to be embedded into the design matrix. We can think of these inputs as the input feature vectors to the NN.

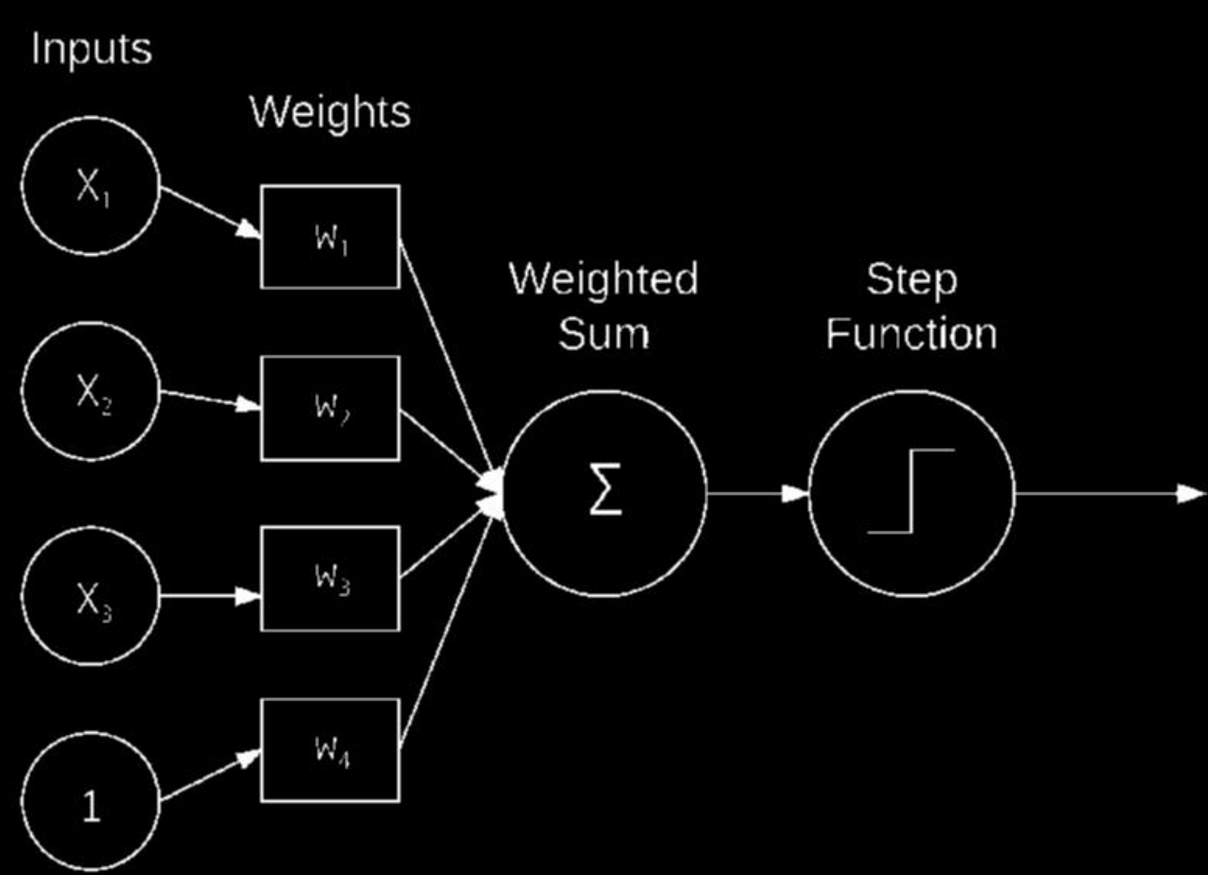

Figure 10.3: A simple NN that takes the weighted sum of the input x and weights w. This weighted sum is then passed through the activation function to determine if the neuron fires.

In practice these inputs could be vectors used to quantify the contents of an image in a systematic, predefined way (e.x., color histograms, Histogram of Oriented Gradients [32], Local Binary Patterns [21], etc.). In the context of deep learning, these inputs are the *raw pixel intensities* of the images themselves.

Each x is connected to a neuron via a weight vector $\boldsymbol{W}$ consists of $w_1, w_2, \ldots, w_n$, meaning that for each input x we also have an associated weight w.

Finally, the *output node* on the right of Figure 10.3 takes the weighted sum, applies an activation function f (used to determine if the neuron "fires" or not), and outputs a value. Expressing the output mathematically, you'll typically encounter the following three forms:

- $f(w_1x_1 + w_2x_2 + \cdots + w_nx_n)$
- $f(\sum_{i=1}^{n} w_ix_i)$
- Or simply, $f(net)$, where $net = \sum_{i=1}^{n} w_ix_i$

Regardless how the output value is expressed, understand that we are simply taking the weighted sum of inputs, followed by applying an activation function f.

Activation Functions

The most simple activation function is the "step function", used by the Perceptron algorithm (which we'll cover in the next section).

$$f(net) = \begin{cases} 1 & \text{if } net > 0 \\ 0 & \text{otherwise} \end{cases}$$

As we can see from the equation above, this is a very simple threshold function. If the weighted sum $\sum_{i=1}^{n} w_ix_i > 0$, we output 1, otherwise, we output 0.

Plotting input values along the x-axis and the output of $f(net)$ along the $y-axis$ we can see why this activation function received its name (Figure 10.4, *top-left*). The output of f is always zero when net is less than or equal zero. If net is greater than to zero, then f will return one. Thus, this function looks like a stair step, not dissimilar to the stairs you walk up and down every day.

However, while being intuitive and easy to use, the step function is not differentiable, which can lead to problems when applying gradient descent and training our network.

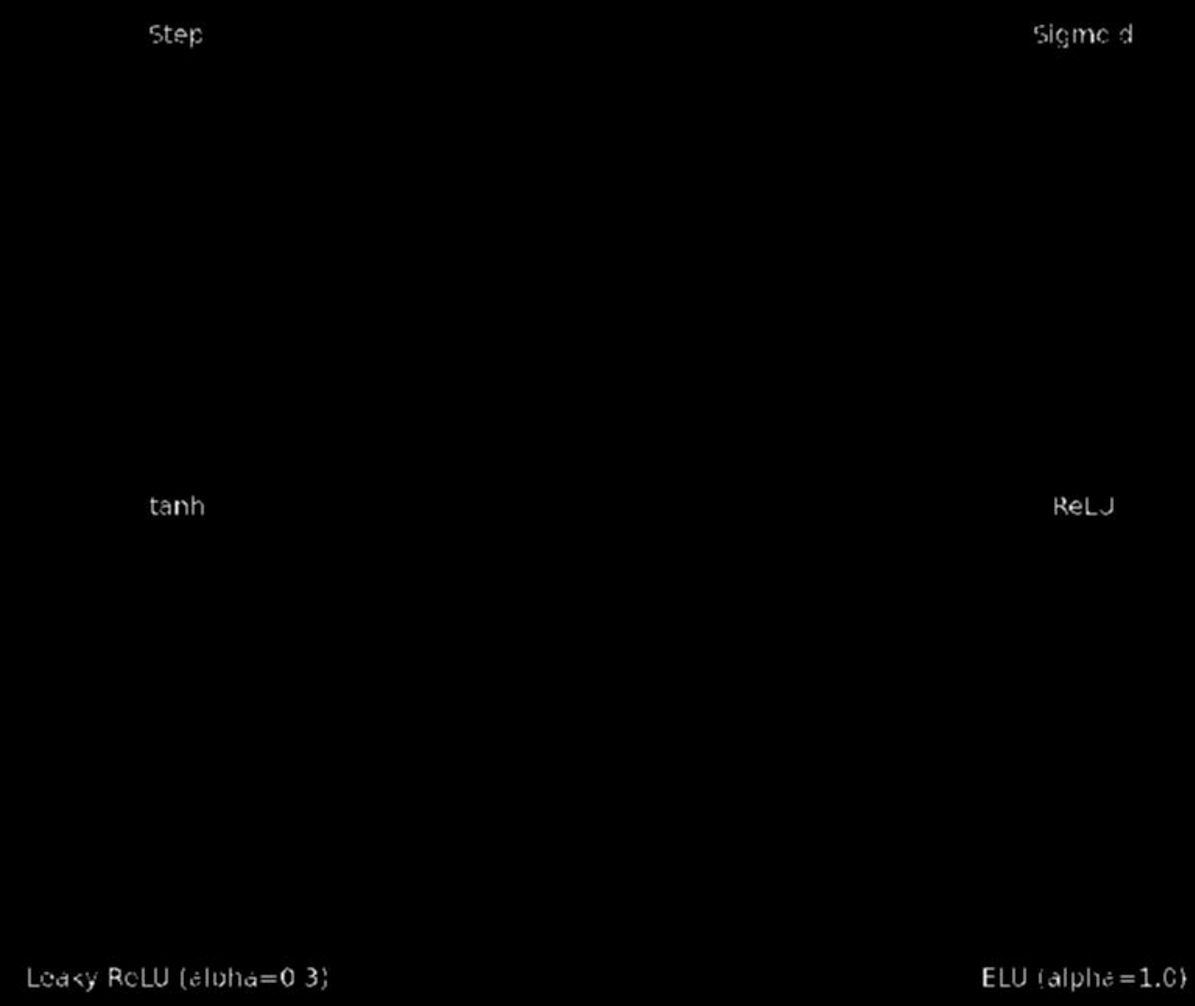

Figure 10.4: **Top-left:** Step function. **Top-right:** Sigmoid activation function. **Mid-left:** Hyperbolic tangent. **Mid-right:** ReLU activation (most used activation function for deep neural networks). **Bottom-left:** Leaky ReLU, variant of the ReLU that allows for negative values. **Bottom-right:** ELU, another variant of ELU that can often perform better than Leaky ReLU.

Instead, a more common activation function used in the history of NN literature is the sigmoid function (Figure 10.4, *top-right*), which follows the equation:

$$t = \sum_{i=1}^{n} w_i x_i \qquad\qquad s(t) = 1/(1+e^{-t}) \tag{10.1}$$

The sigmoid function is a better choice for learning than the simple step function since it:

1. Is continuous and differentiable everywhere.
2. Is symmetric around the y-axis.
3. Asymptotically approaches its saturation values.

The primary advantage here is that the smoothness of the sigmoid function makes it easier to devise learning algorithms. However, there are *two big problems* with the sigmoid function:

1. The outputs of the sigmoid are not zero centered.
2. Saturated neurons essentially kill the gradient, since the delta of the gradient will be extremely small.

The hyperbolic tangent, or *tanh* (with a similar shape of the sigmoid) was also heavily used as an activation function up until the late 1990s (Figure 10.4, *mid-left*): The equation for *tanh* follows:

$$f(z) = tanh(z) = (e^{z} - e^{-z})/(e^{z} - e^{-z}) \tag{10.2}$$

The *tanh* function is zero centered, but the gradients are still killed when neurons become saturated.

We now know there are better choices for activation functions than the sigmoid and *tanh* functions. Specifically, the work of Hahnloser et al. in their 2000 paper, *Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit* [101], introduced the **Rectified Linear Unit (ReLU)**, defined as:

$$f(x) = max(0, x) \tag{10.3}$$

ReLU are also called "ramp functions" due to how they look when plotted (Figure 10.4, *mid-right*). Notice how the function is zero for negative inputs but then linearly increases for positive values. The ReLU function is not saturable and is also extremely computationally efficient.

Empirically, the ReLU activation function tends to outperform *both* the sigmoid and *tanh* functions in nearly all applications. Combined with the work of Hahnloser and Seung in their followup 2003 paper *Permitted and Forbidden Sets in Symmetric Threshold-Linear Networks* [102], it was found that the ReLU activation function has stronger biological motivations than the previous families of an activation functions, including more complete mathematical justifications.

As of 2015, ReLU is the *most popular* activation function used in deep learning [9]. However, a problem arises when we have a value of zero – *the gradient cannot be taken*.

A variant of ReLUs, called *Leaky ReLUs* [103] allow for a small, non-zero gradient when the unit is not active:

$$f(net) = \begin{cases} net & \text{if } net >= 0 \\ \alpha \times net & \text{otherwise} \end{cases}$$

Plotting this function in Figure 10.4 (*bottom-left*), we can see that the function is indeed allowed to take on a negative value, unlike traditional ReLUs which "clamp" the function output at zero.

Parametric ReLUs, or PReLUs for short [96], build on Leaky ReLUs and allow the parameter α to be learned on an activation-by-activation basis, implying that each node in the network can learn a different "coefficient of leakage" separate from the other nodes.

Finally, we also have *Exponential Linear Units (ELUs)* introduced by Clevert et al. in their 2015 paper, *Fast and Accurate Deep Learning by Exponential Linear Units (ELUs)* [104]:

$$f(net) = \begin{cases} net & \text{if } net >= 0 \\ \alpha \times (exp(net) - 1) & \text{otherwise} \end{cases}$$

The value of α is constant and *set when the network architecture is instantiated* – this is unlike PReLUs where α is learned. A typical value for α is $\alpha = 1.0$. Figure 10.4 (*bottom-right*) visualizes the ELU activation function.

Through the work of Clevert et al. (and my own anecdotal experiments), ELUs often obtain higher classification accuracy than ReLUs. ELUs rarely, if ever perform worse than your standard ReLU function.

Which Activation Function Do I Use?

Given the popularity of the most recent incarnation of deep learning, there has been an associated explosion in activation functions. Due to the number of choices of activation functions, both modern (ReLU, Leaky ReLU, ELU, etc.) and "classical" ones (step, sigmoid, *tanh*, etc.), it may appear to be a daunting, perhaps even overwhelming task to select an appropriate activation function.

However, in nearly all situations, I recommend starting with a ReLU to obtain a baseline accuracy (as do most papers published in the deep learning literature). From there you can try swapping out your standard ReLU for a Leaky ReLU variant.

My personal preference is to start with a ReLU, tune my network and optimizer parameters (architecture, learning rate, regularization strength, etc.) and note the accuracy. Once I am reasonably satisfied with the accuracy, I swap in an ELU and often notice a $1-5\%$ improvement in classification accuracy depending on the dataset. Again, this is only my anecdotal advice. You should run your own experiments and note your findings, but as a general rule of thumb, start with a normal ReLU and tune the other parameters in your network – then swap in some of the more "exotic" ReLU variants.

Feedforward Network Architectures

While there are many, *many* different NN architectures, the most common architecture is the *feedforward network*, as presented in Figure 10.5.

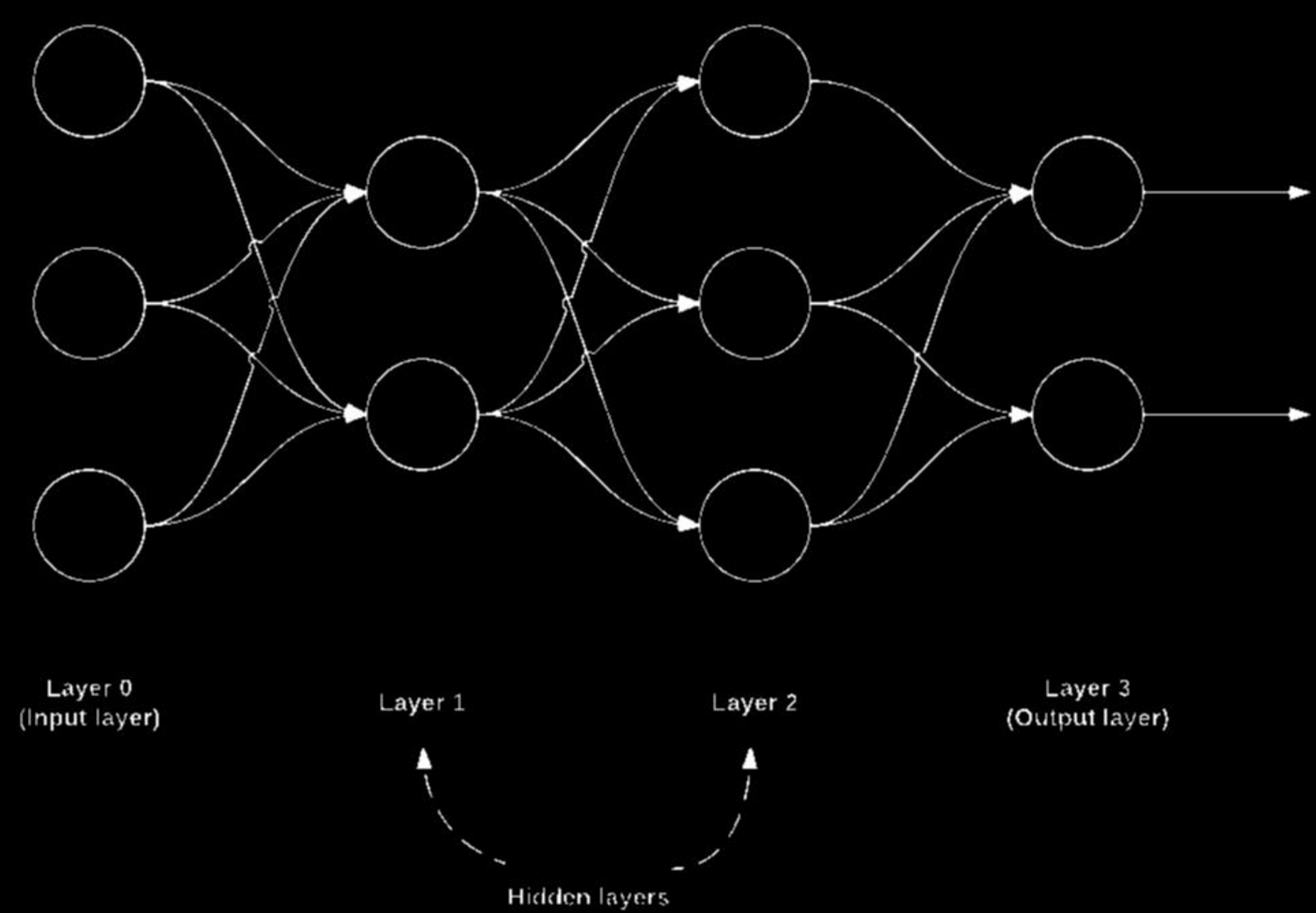

Figure 10.5: An example of a feedforward neural network with 3 input nodes, a hidden layer with 2 nodes a second hidden layer with 3 nodes, and a final output layer with 2 nodes.

In this type of architecture, a connection between nodes is *only allowed* from nodes in layer i to nodes in layer $i+1$ (hence the term, *feedforward*). There are no backward or inter-layer connections allowed. When feedforward networks include *feedback connections* (output connections that feed back into the inputs) they are called **recurrent neural networks**.

In this book we focus on feedforward neural networks as they are the cornerstone of modern deep learning applied to computer vision. As we'll find out in Chapter 11, Convolutional Neural Networks are simply a special case of feedforward neural network.

To describe a feedforward network, we normally use a sequence of integers to quickly and concisely depot the number of nodes in each layer. For example, the network in Figure 10.5 above is a *3-2-3-2* feedforward network:

Layer 0 contains 3 inputs, our x_i values. These could be raw pixel intensities of an image or a feature vector extracted from the image.

Layers 1 and 2 are *hidden layers* containing 2 and 3 nodes, respectively.

Layer 3 is the *output layer or the visible layer* – there is where we obtain the overall output classification from our network. The output layer typically has as many nodes as class labels; one node for each potential output. For example, if we were to build an NN to classify handwritten digits, our output layer would consist of 10 nodes, one for each digit *0-9*.

Neural Learning

Neural learning refers to the method of modifying the weights and connections between nodes in a network. Biologically, we define learning in terms of Hebb's principle:

> *"When an axon of cell A is near enough to excite cell B, and repeatedly or persistently takes place in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased"* – Donald Hebb [105]

In terms of ANNs, this principle implies that there should be an increase in strength of connections between nodes that have similar outputs when presented with the same input. We call this *correlation learning* because the strength of the connections between neurons eventually represents the correlation between outputs.

What are Neural Networks Used For?

Neural Networks can be used in both supervised, unsupervised, and semi-supervised learning tasks, providing the appropriate architecture is used, of course. A complete review of NNs is outside the scope of this book (please see Schmidhuber [40] for an extensive survey of deep artificial networks, along with Mehrota [106] for an review of classical methods); however, common applications of NN include classification, regression, clustering, vector quantization, pattern association, and function approximation, just to name a few.

In fact, for nearly every facet of machine learning, NNs have been applied in some form or another. In the context of this book, we'll be using NNs for *computer vision* and *image classification*.

A Summary of Neural Network Basics

In this section, we reviewed the basics of Artificial Neural Networks (ANNs, or simply NNs). We started by examining the biological motivation behind ANNs, and then learned how we can *mathematically define* a function to mimic the activation of a neuron (i.e., the activation function).

Based on this model of a neuron, we are able to define the *architecture* of a network consisting of (at a bare minimum), an *input layer* and an *output layer*. Some network architectures may include *multiple hidden layers* between the input and output layers. Finally, each layer can have one or more nodes. Nodes in the input layer *do not* contain an activation function (they are "where" the individual pixel intensities of our image are inputted); however, nodes in both the hidden and output layers *do* contain an activation function.

We also reviewed three popular activation functions: *sigmoid*, *tanh*, and *ReLU* (and its variants).

Traditionally the sigmoid and *tanh* functions have been used to train networks; however, since Hahnloser et al.'s 2000 paper [101], the ReLU function has been used more often.

In 2015, ReLU is *by far* the most popular activation function used in deep learning architectures [9]. Based on the success of ReLU, we also have *Leaky ReLUs*, a variant of ReLUs that seek to

improve network performance by allowing the function to take on a negative value. The Leaky ReLU family of functions consists of your standard leaky ReLU variant, PReLUs, and ELUs.

Finally, it's important to note that even though we are focusing on deep learning strictly in the context of *image classification*, neural networks have been used in some fashion in nearly all niches of machine learning.

Now that we understand the basics of NNs, let's make this knowledge more concrete by examining actual architectures and their associated implementations. In the next section, we'll discuss the classic Perceptron algorithm, one of the first ANNs ever to be created.

The Perceptron Algorithm

First introduced by Rosenblatt in 1958, *The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain* [12] is arguably the oldest and most simple of the ANN algorithms. Following this publication, Perceptron-based techniques were all the rage in the neural network community. This paper alone is hugely responsible for the popularity and utility of neural networks today.

But then, in 1969, an "AI Winter" descended on the machine learning community that almost froze out neural networks for good. Minsky and Papert published *Perceptrons: an introduction to computational geometry* [14], a book that effectively stagnated research in neural networks for almost a decade – there is much controversy regarding the book [107], but the authors did successfully demonstrate that a *single layer* Perceptron is unable to separate nonlinear data points.

Given that most real-world datasets are naturally nonlinearly separable, this it seemed that the Perceptron, along with the rest of neural network research, might reach an untimely end.

Between the Minsky and Papert publication and the broken promises of neural networks revolutionizing industry, the interest in neural networks dwindled substantially. It wasn't until we started exploring deeper networks (sometimes called *multi-layer perceptrons*) along with the backpropagation algorithm (Werbos [15] and Rumelhart [16]) that the "AI Winter" in the 1970s ended and neural network research started to heat up again.

All that said, the Perceptron is still a *very important algorithm* to understand as it sets the stage for more advanced multi-layer networks. We'll start this section with a review of the Perceptron architecture and explain the training produced (called the **delta rule**) used to train the Perceptron. We'll also look at the *termination criteria* of the network (i.e., when the Perceptron should stop training). Finally, we'll implement the Perceptron algorithm in pure Python and use it to study and examine how the network is unable to learn nonlinearly separable datasets.

AND, OR, and XOR Datasets

Before we study the Perceptron itself, let's first discuss "bitwise operations", including AND, OR, and XOR (exclusive OR). If you've taken an introductory level computer science course before you might already be familiar with bitwise functions.

Bitwise operators and associated bitwise datasets accept two input bits and produce a final output bit after applying the operation. Given two input bits, each potentially taking on a value of 0 or 1, there are four possible combinations of of these two bits – Table 10.1 provides the possible input and output values for AND, OR, and XOR:

As we can see on the *left*, a logical AND is true *if and only if* both input values are 1. If *either* of the input values are 0, the AND returns 0. Thus, there is only one combination, $x_0 = 1$ and $x_1 = 1$ when the output of AND is true.

In the *middle*, we have the OR operation which is true when only *one* of the input values is 1. Thus, there are three possible combinations of the two bits x_0 and x_1 that produce a value of $y = 1$.

Finally, the *right* displays the XOR operation which is true *if and only if* one if the inputs is 1 *but not both*. While OR had three possible situations where $y = 1$, XOR only has two.

x_0	x_1	$x_0 \& x_1$
0	0	0
0	1	0
1	0	0
1	1	1

x_0	x_1	$x_0 \mid x_1$
0	0	0
0	1	1
1	0	1
1	1	1

x_0	x_1	$x_0 \wedge x_1$
0	0	0
0	1	1
1	0	1
1	1	0

Table 10.1: **Left:** The bitwise AND dataset. Given two inputs, the output is only 1 if both inputs are 1. **Middle:** The bitwise OR dataset. Given two inputs, the output is 1 if *either* of the two inputs is 1. **Right:** The XOR (e(X)clusive OR) dataset. Given two inputs, the output 1 if and only if one if the inputs is 1, *but not both*.

We often use these simple "bitwise datasets" to test and debug machine learning algorithms. If we plot and visualize the AND, OR, and XOR values (with red circles being zero outputs and blue stars one outputs) in Figure 10.6, you'll notice an interesting pattern:

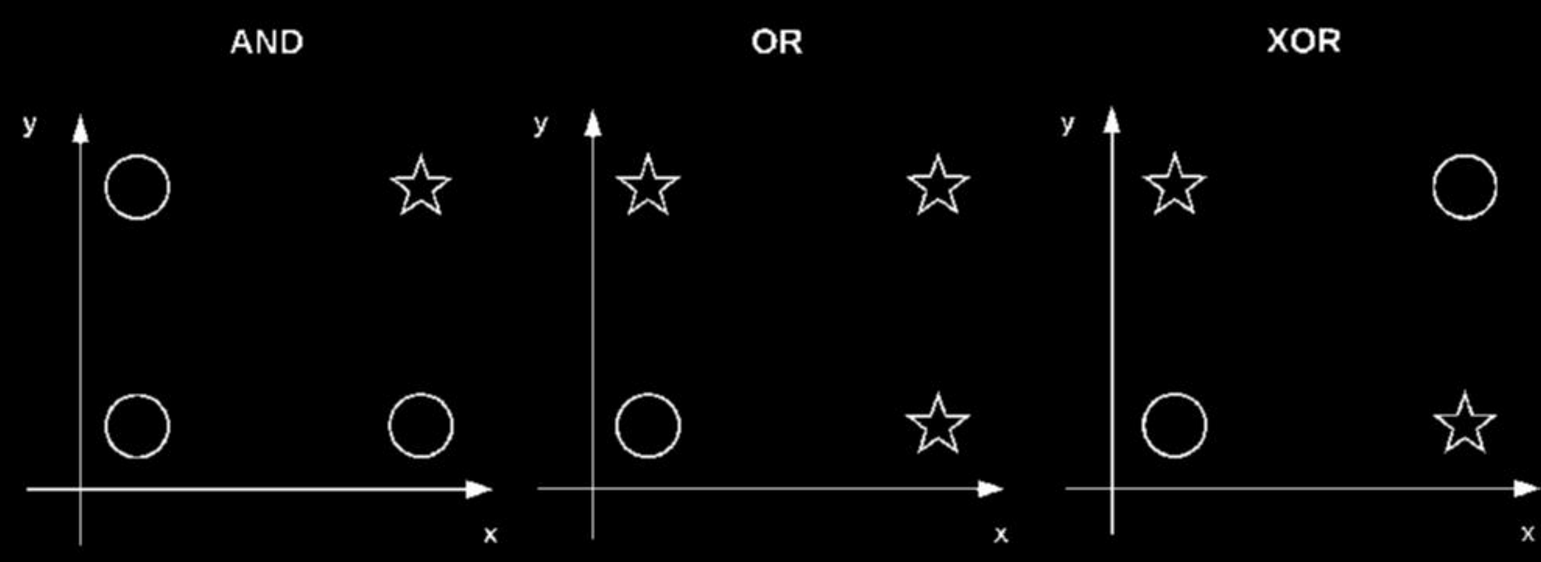

Figure 10.6: Both the AND and OR bitwise datasets are linearly separable, meaning that we can draw a *single line* (green) that separates the two classes. However, for XOR it is impossible to draw a single line that separates the two classes – this is therefore a nonlinearly separable dataset.

Both AND and OR are linearly separable – we can clearly draw a line that separates the 0 and 1 classes – the same is not true for XOR. Take the time now to convince yourself that it is *not possible* to draw a line that cleanly separates the two classes in the XOR problem. XOR is, therefore, an example of a *nonlinearly separable* dataset.

Ideally, we would like our machine learning algorithms to be able to separate nonlinear classes as most datasets encountered in the real-world are nonlinear. Therefore, when constructing, debugging, and evaluating a given machine learning algorithm, we may use the bitwise values x_0 and x_1 as our *design matrix* and then try to predict the corresponding y values.

Unlike our standard procedure of splitting our data into *training* and *testing* splits, when using bitwise datasets we simply train and evaluate our network on the same set of data. Our goal here is simply to determine if it's even *possible* for our learning algorithm to learn the patterns in the data. As we'll find out, the Perceptron algorithm can correctly classify the AND and OR functions but fails to classify the XOR data.

Perceptron Architecture

Rosenblatt [12] defined a Perceptron as a system that learns using labeled examples (i.e., supervised learning) of feature vectors (or raw pixel intensities), mapping these inputs to their corresponding output class labels.

In its simplest form, a Perceptron contains N input nodes, one for each entry in the *input row* of

the design matrix, followed by *only one layer* in the network with just a *single node* in that layer (Figure 10.7).

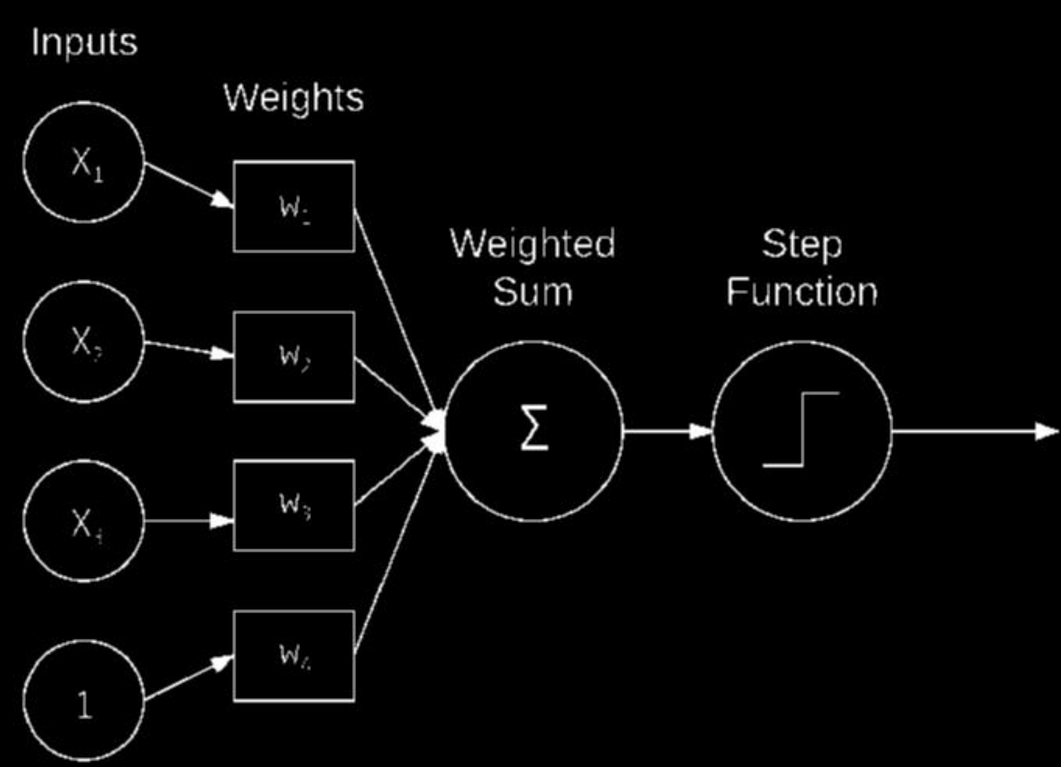

Figure 10.7: Architecture of the Perceptron network.

There exist connections and their corresponding weights $w_1, w_2, \ldots, w_i$ from the input x_i's to the single output node in the network. This node takes the weighted sum of inputs and applies a *step function* to determine the output class label. The Perceptron outputs either a 0 or a 1 – 0 for class #1 and 1 for class #2; thus, in its original form, the Perceptron is simply a binary, two class classifier.

Perceptron Training Procedure and the Delta Rule

Training a Perceptron is a fairly straightforward operation. Our goal is to obtain a set of weights $\boldsymbol{w}$ that accurately classifies each instance in our training set. In order to train our Perceptron, we iteratively feed the network our training data *multiple times*. Each time the network has seen the *full set* of training data, we say an *epoch* has passed. It normally takes many epochs until a weight vector $\boldsymbol{w}$ can be learned to linearly separate our two classes of data.

The pseudocode for the Perceptron training algorithm can be found below:

The actual "learning" takes place in Steps 2b and 2c. First, we pass the feature vector x_j through the network, take the dot product weight the weights $\boldsymbol{w}$ and obtain the output y_j. This value is then passed through the step function which will return 1 if $x > 0$ and 0 otherwise.

Now we need to update our weight vector $\boldsymbol{w}$ to step in the direction that is "closer" to the correct classification. This update of the weight vector is handled by the *delta rule* in Step 2c.

The expression $(d_j - y_j)$ determines if the output classification is correct or not. If the classification is *correct*, then this difference will be zero. Otherwise, the difference will be either positive or negative, giving us the direction in which our weights will be updated (ultimately bringing us closer to the correct classification). We then multiply $(d_j - y_j)$ by x_j, moving us closer to the correct classification.

1. Initialize our weight vector $\boldsymbol{w}$ with small random values
2. Until Perceptron converges:
   (a) Loop over each feature vector x_j and true class label d_j in our training set D
   (b) Take $\boldsymbol{x}$ and pass it through the network, calculating the output value: $y_j = f(\boldsymbol{w}(t) \cdot \boldsymbol{x}_j)$
   (c) Update the weights $\boldsymbol{w}$: $w_i(t+1) = w_i(t) - \eta(d_j - y_j)x_{j,i}$ for all features $0 <= i <= n$

Figure 10.8: The Perceptron algorithm training procedure.

The value α is our *learning rate* and controls how large (or small) of a step we take. It's *critical* that this value is set correctly. A larger value of α will cause us to take a step in the right direction; however, this step could be *too large*, and we could easily overstep a local/global optimum.

Conversely, a small value of α allows us to take tiny baby steps in the right direction, ensuring we don't overstep a local/global minimum; however, these tiny baby steps make take an intractable amount of time for our learning to converge.

Finally, we add in the previous weight vector at time t, $\boldsymbol{w}_{j(t)}$ which completes the process of "stepping" towards the correct classification. If you find this training procedure a bit confusing, don't worry – we'll be covering it in detail with Python code later in Section 10.1.2.

Perceptron Training Termination

The Perceptron training process is allowed to proceed until all training samples are classified correctly *or* a preset number of epochs is reached. Termination is ensured if α is sufficiently small *and* the training data is linearly separable.

So, what happens if our data is not linearly separable or we make a poor choice in α? Will training continue infinitely? In this case, no – we normally stop after a set number of epochs has been hit or if the number of misclassifications has not changed in a large number of epochs (indicating that the data is not linearly separable). For more details on the perceptron algorithm, please refer to either Andrew Ng's Stanford lecture [76] or the introductory chapters of Mehrota et al. [106].

Implementing the Perceptron in Python

Now that we have studied the Perceptron algorithm, let's implement the actual algorithm in Python. Create a file named `perceptron.py` in your `pyimagesearch.nn` package – this file will store our actual `Perceptron` implementation:

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

After you've created the file, open it up, and insert the following code:

```
1
2         numpy    np
3
4        Perceptron:
5                 (    , N, alpha=   ):
6
7                 .W = np.random.randn(N +  ) / np.sqrt(N)
8                 .alpha = alpha
```

Line 5 defines the constructor to our `Perceptron` class, which accepts a single required parameter followed by a second optional one:

1. `N`: The number of columns in our input feature vectors. In the context of our bitwise datasets, we'll set `N` equal to two since there are two inputs.
2. `alpha`: Our learning rate for the Perceptron algorithm. We'll set this value to `0.01` by default. Common choices of learning rates are normally in the range $\alpha = 0.1, 0.01, 0.001$.

Line 7 files our weight matrix `W` with random values sampled from a "normal" (Gaussian) distribution with zero mean and unit variance. The weight matrix will have $N + 1$ entries, one for each of the `N` inputs in the feature vector, plus one for the bias. We divide `W` by the square-root of the number of inputs, a common technique used to scale our weight matrix, leading to faster convergence. We will cover weight initialization techniques later in this chapter.

Next, let's define the `step` function:

```
10                  (     , x):
11
12                        x >
```

This function mimics the behavior of the step equation in Section 10.4 above – if `x` is positive we return `1`, otherwise we return `0`.

To actually train the Perceptron we'll define a function named `fit`. If you have any previous experience with machine learning, Python, and the scikit-learn library then you'll know that it's common to name your training procedure function `fit`, as in *"fit a model to the data"*:

```
14                (     , X, y, epochs=  ):
15
16
17
18            X = np.c_[X, np.ones((X.shape[ ]))]
```

The `fit` method requires two parameters followed by a single optional one:

The `X` value is our actual training data. The `y` variable is our target output class labels (i.e., what our network *should* be predicting). Finally, we supply `epochs`, the number of epochs our Perceptron will train for.

Line 18 applies the bias trick (Section 9.3) by inserting a column of ones into the training data, which allows us to treat the bias as a trainable parameter *directly* inside the weight matrix.

Next, let's review the actual training procedure:

```
20
21                epoch    np.arange( , epochs):
22
23                    (x, target)       (X, y):
24
25
26
27                     p =     .step(np.dot(x,     .W))
28
29
30
31                          p != target:
32
33                            error = p - target
34
35
36                                .W += -    .alpha * error * x
```

On **Line 21** we start looping over the desired number of `epochs`. For each epoch, we also loop over each individual data point `x` and output `target` class label (**Line 23**).

Line 27 takes the dot product between the input features `x` and the weight matrix `W`, then passes the output through the `step` function to obtain the prediction by the Perceptron.

Applying the same training procedure detailed in Listing 10.8 above, we only perform a weight update if our prediction *does not* match the target (**Line 31**). If this is the case, we determine the `error` (**Line 33**) by computing the sign (either positive or negative) via the difference operation.

Updating the weight matrix is handled on **Line 36** where we take a step towards the correct classification, scaling this step by our learning rate `alpha`. Over a series of epochs, our Perceptron is able to learn patterns in the underlying data and shift the values of the weight matrix such that we correctly classify our input samples `x`.

The last function we need to define is `predict`, which, as the name suggests, is used to *predict* the class labels for a given set of input data:

```
38                 (     , X, addBias=    ):
39
40             X = np.atleast_2d(X)
41
42
43                 addBias:
44
45
46                   X = np.c_[X, np.ones((X.shape[ ]))]
47
48
49
50
51                         .step(np.dot(X,      .W))
```

Our `predict` method requires a set of input data `X` that needs to be classified. A check on **Line 43** is made to see if a bias column needs to be added.

Obtaining the output predictions for `X` is the same as the training procedure – simply take the dot product between the input features `X` and our weight matrix `W`, and then pass the value through our step function. The output of the step function is returned to the calling function.

Now that we've implemented our `Perceptron` class, let's try to apply it to our bitwise datasets and see how the neural network performs.

Evaluating the Perceptron Bitwise Datasets

To start, let's create a file named `perceptron_or.py` that attempts to fit a Perceptron model to the bitwise OR dataset:

```
1
2         pyimagesearch.nn         Perceptron
3         numpy    np
4
5
6   X = np.array([[ ,  ], [ ,  ], [ ,  ], [ ,  ]])
7   y = np.array([[ ], [ ], [ ], [ ]])
8
9
10       ("[INFO] training perceptron...")
11  p = Perceptron(X.shape[ ], alpha=   )
12  p.fit(X, y, epochs=  )
```

Lines 2 and 3 import our required Python packages. We'll be using our `Perceptron` implementation from Section 10.1.2 above. **Lines 6 and 7** define the OR dataset based on Table 10.1.

Lines 11 and 12 train our Perceptron with a learning rate of $\alpha = 0.1$ for a total of 20 epochs.

We can then evaluate our Perceptron on the data to validate that it did, in fact, learn the OR function:

```
14
15      ("[INFO] testing perceptron...")
16
17
18    (x, target)      (X, y):
19
20
21     pred = p.predict(x)
22          ("[INFO] data={}, ground truth={}, pred={}".format(
23          x, target[ ], pred))
```

On **Line 18** we loop over each of the data points in the OR dataset. For each of these data points, we pass it through the network and obtain the prediction (**Line 21**).

Finally, **Lines 22 and 23** display the input data point, the ground-truth label, as well as our predicted label to our console.

To see if our Perceptron algorithm is able to learn the OR function, just execute the following command:

```
python perceptron_or.py
```

Sure enough, our neural network is able to correctly predict that the OR operation for $x_0 = 0$ and $x_1 = 0$ is zero – all other combinations are one.

Now, let's move on to the AND function – create a new file named `perceptron_and.py` and insert the following code:

```
1
2       pyimagesearch.nn        Perceptron
3         numpy    np
4
5
6  X = np.array([[ ,  ], [ ,  ], [ ,  ], [ ,  ]])
7  y = np.array([[ ], [ ], [ ], [ ]])
8
9
10      ("[INFO] training perceptron...")
11 p = Perceptron(X.shape[ ], alpha=   )
12 p.fit(X, y, epochs=  )
13
```

```
14
15      ("[INFO] testing perceptron...")
16
17
18     (x, target)      (X, y):
19
20
21      pred = p.predict(x)
22           ("[INFO] data={}, ground truth={}, pred={}".format(
23           x, target[ ], pred))
```

Notice here that the *only* lines of code that have changed are **Lines 6 and 7** where we define the AND dataset rather than the OR dataset.

Executing the following command, we can evaluate the Perceptron on the AND function:

```
python perceptron_and.py
```

Again, our Perceptron was able to correctly model the function. The AND function is only true when *both* $x_0 = 1$ and $x_1 = 1$ – for all other combinations the bitwise AND is zero.

Finally, let's take a look at the nonlinearly separable XOR function inside `perceptron_xor.py`:

```
1
2      pyimagesearch.nn        Perceptron
3        numpy    np
4
5
6  X = np.array([[ ,  ], [ ,  ], [ ,  ], [ ,  ]])
7  y = np.array([[ ], [ ], [ ], [ ]])
8
9
10     ("[INFO] training perceptron...")
11 p = Perceptron(X.shape[ ], alpha=   )
12 p.fit(X, y, epochs=  )
13
14
15     ("[INFO] testing perceptron...")
16
17
18    (x, target)      (X, y):
19
20
21     pred = p.predict(x)
22          ("[INFO] data={}, ground truth={}, pred={}".format(
23          x, target[ ], pred))
```

Again, the only lines of code that have been changed are **Lines 6 and 7** where we define the XOR data. The XOR operator is true *if and only if* one (but not both) x's are one.

Executing the following command we can see that the Perceptron *cannot* learn this nonlinear relationship:

```
python perceptron_xor.py
```

No matter how many times you run this experiment with varying learning rates or different weight initialization schemes, you will *never* be able to correctly model the XOR function with a single layer Perceptron. Instead, what we need is *more layers* – and with that, comes the start of deep learning.

Backpropagation and Multi-layer Networks

Backpropagation is arguably *the* most important algorithm in neural network history – without (efficient) backpropagation, it would be *impossible* to train deep learning networks to the depths that we see today. Backpropagation can be considered the cornerstone of modern neural networks and deep learning.

The original incarnation of backpropagation was introduced back in the 1970s, but it wasn't until the seminal 1986 paper, *Learning representations by back-propagating errors* by Rumelhart, Hinton, and Williams [16], were we able to devise a faster algorithm, more adept to training deeper networks.

There are quite literally hundreds (if not thousands) of tutorials on backpropagation available today. Some of my favorites include:

1. Andrew Ng's discussion on backpropagation inside the Machine Learning course by Coursera [76].
2. The heavily mathematically motivated Chapter 2 – *How the backpropagation algorithm works* from *Neural Networks and Deep Learning* by Michael Nielsen [108].
3. Stanford's cs231n exploration and analysis of backpropagation [57].
4. Matt Mazur's excellent concrete example (with actual worked numbers) that demonstrate how backpropagation works [109].

As you can see, there are no shortage of backpropagation guides – instead of regurgitating and reiterating what has been said by others hundreds of times before, I'm going to take a different approach and do what makes PyImageSearch publications special:

Construct an intuitive, easy to follow implementation of the backpropagation algorithm using the Python language.

Inside this implementation, we'll build an actual neural network and train it using the backpropagation algorithm. By the time you finish this section, you'll understand how backpropagation works – and perhaps more importantly, you'll have a stronger understanding of how this algorithm is used to train neural networks from scratch.

Backpropagation

The backpropagation algorithm consists of two phases:

1. The *forward pass* where our inputs are passed through the network and output predictions obtained (also known as the *propagation* phase).

x_0	x_1	y
0	0	0
0	1	1
1	0	1
1	1	0

x_0	x_1	x_2
0	0	1
0	1	1
1	0	1
1	1	1

Table 10.2: **Left:** The bitwise XOR dataset (including class labels). **Right:** The XOR dataset design matrix with a bias column inserted (excluding class labels for brevity).

2. The *backward pass* where we compute the gradient of the loss function at the final layer (i.e., predictions layer) of the network and use this gradient to recursively apply the chain rule to update the weights in our network (also known as the *weight update* phase).

We'll start by reviewing each of these phases at a high level. From there, we'll implement the backpropagation algorithm using Python. Once we have implemented backpropagation we'll want to be able to make *predictions* using our network – this is simply the forward pass phase, only with a small adjustment (in terms of code) to make the predictions more efficient.

Finally, I'll demonstrate how to train a custom neural network using backpropagation and Python on both the:

1. XOR dataset
2. MNIST dataset

The Forward Pass

The purpose of the forward pass is to propagate our inputs through the network by applying a series of dot products and activations until we reach the output layer of the network (i.e., our predictions). To visualize this process, let's first consider the XOR dataset (Table 10.2, *left*).

Here we can see that each entry X in the design matrix (left) is 2-dim – each data point is represented by *two* numbers. For example, the first data point is represented by the feature vector $(0, 0)$, the second data point by $(0, 1)$, etc. We then have our output values y as the right column. Our target output values are the *class labels*. Given an input from the design matrix, our goal is to correctly predict the target output value.

As we'll find out in Section 10.1.3 below, to obtain perfect classification accuracy on this problem we'll need a feedforward neural network with at least a single hidden layer, so let's go ahead and start with a $2 - 2 - 1$ architecture (Figure 10.9, *top*). This is a good start; however, we're forgetting to include the bias term. As we know from Chapter 9, there are two ways to include the bias term $\boldsymbol{b}$ in our network. We can either:

1. Use a separate variable.
2. Treat the bias as a trainable parameter *within* the weight matrix by inserting a column of 1's into the feature vectors.

Inserting a column of 1's into our feature vector is done programmatically, but to ensure we understand this point, let's update our XOR design matrix to explicitly see this taking place (Table 10.2, *right*). As you can see, a column of 1's have been added to our feature vectors. In practice you can insert this column anywhere you like, but we typically place it either as (1) the first entry in the feature vector or (2) the last entry in the feature vector.

Since we have changed the size of our input feature vector (normally performed *inside* neural network implementation itself so that we do not need to explicitly modify our design matrix), that changes our (perceived) network architecture from $2 - 2 - 1$ to an (internal) $3 - 3 - 1$ (Figure 10.9, *bottom*).

We'll still refer to this network architecture as $2 - 2 - 1$, but when it comes to implementation, it's actually $3 - 3 - 1$ due to the addition of the bias term embedded in the weight matrix.

Companion Website

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

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

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

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

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

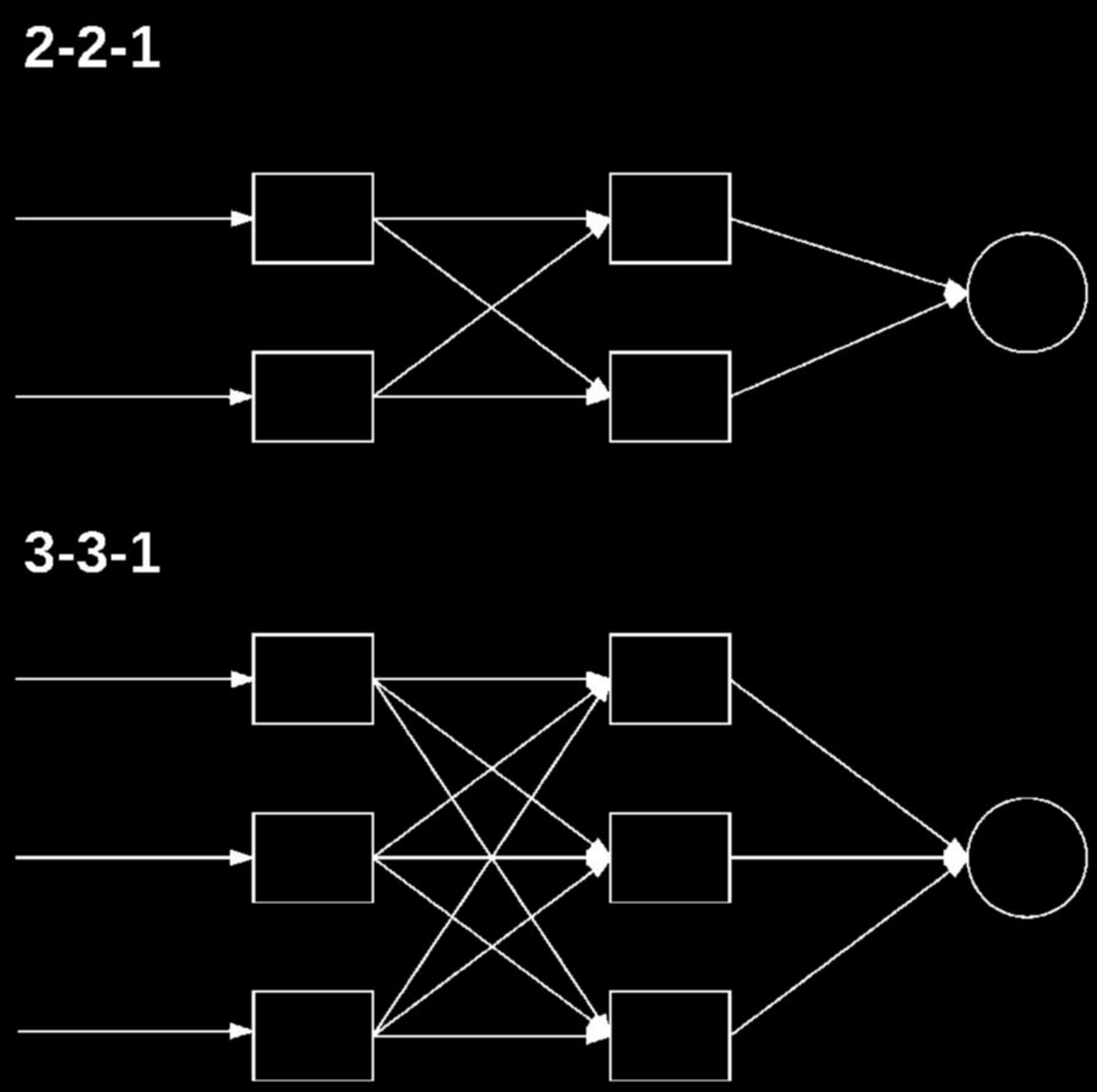

Figure 10.9: **Top:** To build a neural network to correctly classify the XOR dataset, we'll need a network with two input nodes, two hidden nodes, and one output node. This gives rise to a $2-2-1$ architecture. **Bottom:** Our actual internal network architecture representation is $3-3-1$ due to the bias trick. In the vast majority of neural network implementations this adjustment to the weight matrix happens internally and is something that you do not need to worry about; however, it's still important to understand what is going on under the hood.

Finally, recall that both our input layer and all hidden layers require a bias term; however, the final output layer *does not* require a bias. The benefit of applying the bias trick is that we do not need to explicitly keep track of the bias parameter any longer – it is now a trainable parameter *within* the weight matrix, thus making training more efficient and substantially easier to implement. Please see Chapter 9 for a more thorough discussion on why this bias trick works.

To see the forward pass in action, we first initialize the weights in our network, as in Figure 10.10. Notice how each arrow in the weight matrix has a value associated with it – this is the *current* weight value for a given node and signifies the amount in which a given input is amplified or diminished. This weight value will then be *updated* during the backpropagation phase.

On the far left of Figure 10.10, we present the feature vector $(0, 1, 1)$ (and target output value 1 to the network). Here we can see that 0, 1, and 1 have been assigned to the three input nodes in the network. To propagate the values through the network and obtain the final classification, we need to take the dot product between the inputs and the weight values, followed by applying an activation function (in this case, the *sigmoid* function, σ).

Let's compute the inputs to the three nodes in the hidden layers:

1. $\sigma((0 \times 0.351) + (1 \times 1.076) + (1 \times 1.116)) = 0.899$

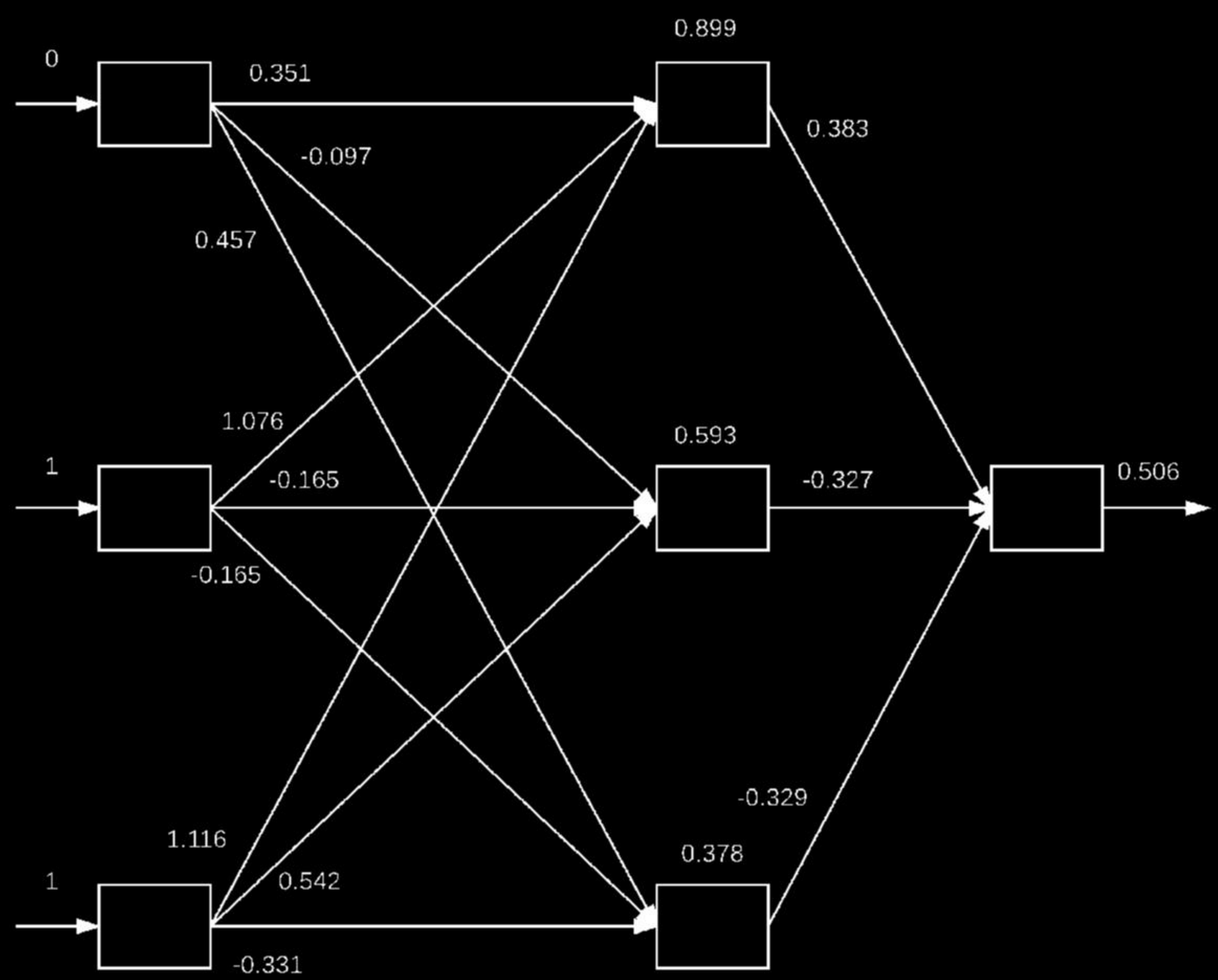

Figure 10.10: An example of the forward propagation pass. The input vector $[0, 1, 1]$ is presented to the network. The dot product between the inputs and weights are taken, followed by applying the sigmoid activation function to obtain the values in the hidden layer (0.899, 0.593, and 0.378, respectively). Finally, the dot product and sigmoid activation function is computed for the final layer, yielding an output of 0.506. Applying the step function to 0.506 yields 1, which is indeed the correct target class label.

2. $\sigma((0 \times -0.097) + (1 \times -0.165) + (1 \times 0.542)) = 0.593$
3. $\sigma((0 \times 0.457) + (1 \times -0.165) - (1 \times -0.331)) = 0.378$

Looking at the node values of the hidden layers (Figure 10.10, *middle*), we can see the nodes have been updated to reflect our computation.

We now have our *inputs* to the hidden layer nodes. To compute the output prediction, we once again compute the dot product followed by a sigmoid activation:

$$\sigma((0.899 \times 0.383) + (0.593 \times -0.327) + (0.378 \times -0.329)) = 0.506 \tag{10.4}$$

The output of the network is thus 0.506. We can apply a step function to determine if this output is the correct classification or not:

$$f(net) = \begin{cases} 1 & \text{if } net > 0 \\ 0 & \text{otherwise} \end{cases}$$

Applying the step function with $net = 0.506$ we see that our network predicts 1 which is, in fact, the correct class label. However, our network is *not very confident* in this class label – the predicted value 0.506 is very close to the threshold of the step. Ideally, this prediction should be closer to $0.98 - 0.99$., implying that our network has truly learned the underlying pattern in the dataset. In order for our network to actually "learn", we need to apply the backward pass.

The Backward Pass

In order to apply the backpropagation algorithm, our activation function must be *differentiable* so that we can compute the *partial derivative* of the error with respect to a given weight $w_{i,j}$, loss (E), node output o_j, and network output net_j.

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{i,j}} \tag{10.5}$$

As the calculus behind backpropagation has been exhaustively explained many times in previous works (see Andrew Ng [76], Michael Nielsen [108], Matt Mazur [109]), I'm going to skip the derivation of the backpropagation chain rule update and instead explain it via code in the following section.

For the mathematically astute, please see the references above for more information on the chain rule and its role in the backpropagation algorithm. By explaining this process in code, my goal is to help readers understand backpropagation through a more intuitive, implementation sense.

Implementing Backpropagation with Python

Let's go ahead and get started implementing backpropagation. Open up a new file, name it `neuralnetwork.py`, and let's get to work:

```
1
2          numpy    np
3
4         NeuralNetwork:
5                   (    , layers, alpha=   ):
6
7
8                 .W = []
9                 .layers = layers
10                .alpha = alpha
```

On **Line 2** we import the only required package we'll need for our implementation of backpropagation – the NumPy numerical processing library.

Line 5 then defines the constructor to our `NeuralNetwork` class. The constructor requires a single argument, followed a by a second optional one:

- `layers`: A list of integers which represents the actual *architecture* of the feedforward network. For example, a value of $[2, 2, 1]$ would imply that our first input layer has two nodes, our hidden layer has two nodes, and our final output layer has one node.
- `alpha`: Here we can specify the learning rate of our neural network. This value is applied during the weight update phase.

Line 8 initializes our list of weights for each layer, `W`. We then store `layers` and `alpha` on **Lines 9 and 10**.

Our weights list `W` is empty, so let's go ahead and initialize it now:

```
12
13
14          i    np.arange( ,    (layers) -  ):
15
16
17
18           w = np.random.randn(layers[i] +  , layers[i +  ] +  )
19               .W.append(w / np.sqrt(layers[i]))
```

On **Line 14** we start looping over the number of layers in the network (i.e., `len(layers)`), but we stop before the final two layer (we'll find out exactly why later in the explantation of this constructor).

Each layer in the network is randomly initialized by constructing an MxN weight matrix by sampling values from a standard, normal distribution (**Line 18**). The matrix is MxN since we wish to connect every node in *current layer* to every node in the *next layer*.

For example, let's suppose that `layers[i] = 2` and `layers[i + 1] = 2`. Our weight matrix would, therefore, be $2x2$ to connect all sets of nodes between the layers. However, we need to be careful here, as we are forgetting an important component – *the bias term*. To account for the bias, we add one to the number of `layers[i]` and `layers[i + 1]` – doing so changes our weight matrix `w` to have the shape $3x3$ given $2 + 1$ nodes for the current layer and $2 + 1$ nodes for the next layer. We scale `w` by dividing by the square root of the number of nodes in the current layer, thereby normalizing the variance of each neuron's output [57] (**Line 19**).

The final code block of the constructor handles the special where the input connections need a bias term, but the output does not:

```
21
22
23          w = np.random.randn(layers[- ] +  , layers[- ])
24              .W.append(w / np.sqrt(layers[- ]))
```

Again, these weight values are randomly sampled and then normalized.

The next function we define is a Python "magic method" named `__repr__` – this function is useful for debugging:

```
26                  (    ):
27
28
29                  "NeuralNetwork: {}".format(
30                " ".join(   (l)     l        .layers))
```

In our case, we'll format a string for our `NeuralNetwork` object by concatenating the integer value of the number of nodes in each layer. Given a `layers` value of `(2, 2, 1)`, the output of calling this function will be:

```
1  >>>      pyimagesearch.nn        NeuralNetwork
2  >>> nn = NeuralNetwork([ ,  ,  ])
3  >>>      (nn)
4  NeuralNetwork:  - -
```

Next, we can define our sigmoid activation function:

```
32                  (     , x):
33
34
35                        / (   + np.exp(-x))
```

As well as the *derivative* of the sigmoid which we'll use during the backward pass:

```
37                          (     , x):
38
39
40
41                  x * (   - x)
```

Again, note that whenever you perform backpropagation, you'll always want to choose an activation function that is *differentiable*.

We'll draw inspiration from the scikit-learn library and define a function named `fit` which will be responsible for actually training our `NeuralNetwork`:

```
43                (     , X, y, epochs=     , displayUpdate=    ):
44
45
46
47          X = np.c_[X, np.ones((X.shape[ ]))]
48
49
50              epoch    np.arange( , epochs):
51
52
53                  (x, target)       (X, y):
54                       .fit_partial(x, target)
55
56
57                  epoch ==      (epoch +   ) % displayUpdate ==   :
58                    loss =     .calculate_loss(X, y)
59                         ("[INFO] epoch={}, loss={:.7f}".format(
60                         epoch +   , loss))
```

The `fit` method requires two parameters, followed by two optional ones. The first, `X`, is our *training data*. The second, `y`, is the corresponding class labels for each entry in `X`. We then specify `epochs`, which is the number of epochs we'll train our network for. The `displayUpdate` parameter simply controls how many N epochs we'll print training progress to our terminal.

On **Line 47** we perform the bias trick by inserting a column of 1's as the last entry in our feature matrix, `X`. From there, we start looping over our number of `epochs` on **Line 50**. For each epoch, we'll loop over each individual data point in our training set, make a prediction on the data point, compute the backpropagation phase, and then update our weight matrix (**Lines 53 and 54**). **Lines 57-60** simply check to see if we should display a training update to our terminal.

The actual heart of the backpropagation algorithm is found inside our `fit_partial` method below:

```
62                      (     , x, y):
63
64
65
66
67          A = [np.atleast_2d(x)]
```

The `fit_partial` function requires two parameters:

- x: An individual data point from our design matrix.
- y: The corresponding class label.

We then initialize a list, `A`, on **Line 67** – this list is responsible for storing the output activations for each layer as our data point x forward propagates through the network. We initialize this list with x, which is simply the input data point.

From here, we can start the forward propagation phase:

```
69
70
71              layer    np.arange( ,    (    .W)):
72
73
74
75
76               net = A[layer].dot(    .W[layer])
77
78
79
80               out =     .sigmoid(net)
81
82
83
84               A.append(out)
```

We start looping over every layer in the network on **Line 71**. The *net input* to the current `layer` is computed by taking the dot product between the activation and the weight matrix (**Line 76**). The *net output* of the current layer is then computed by passing the net input through the nonlinear sigmoid activation function. Once we have the net output, we add it to our list of activations (**Line 84**).

Believe it or not, this code is the *entirety* of the forward pass described in Section 10.1.3 above – we are simply looping over each of the layers in the network, taking the dot product between the activation and the weights, passing the value through a nonlinear activation function, and continuing to the next layer. The final entry in `A` is thus the output of the last layer in our network (i.e., the *prediction*).

Now that the forward pass is done, we can move on to the slightly more complicated backward pass:

```
86
87
88
89
90
```

```
91          error = A[- ] - y
92
93
94
95
96
97          D = [error *     .sigmoid_deriv(A[- ])]
```

The first phase of the backward pass is to compute our `error`, or simply the difference between our *predicted* label and the *ground-truth* label (**Line 91**). Since the final entry in the activations list `A` contains the output of the network, we can access the output prediction via `A[-1]`. The value `y` is the target output for the input data point `x`.

> When using the Python programming language, specifying an index value of `-1` indicates that we would like to access the *last* entry in the list. You can read more about Python array indexes and slices in this tutorial: .

Next, we need to start applying the chain rule to build our list of deltas, `D`. The deltas will be used to update our weight matrices, scaled by the learning rate `alpha`. The first entry in the deltas list is the error of our output layer multiplied by the derivative of the sigmoid for the output value (**Line 97**).

Given the delta for the final layer in the network, we can now work backward using `for` loop:

```
99
100
101
102
103             layer    np.arange(   (A) -  ,  , - ):
104
105
106
107
108
109              delta = D[- ].dot(    .W[layer].T)
110              delta = delta *     .sigmoid_deriv(A[layer])
111              D.append(delta)
```

On **Line 103** we start looping over each of the layers in the network (ignoring the previous two layers as they are already accounted for in **Line 97**) in *reverse order* as we need to work *backward* to compute the delta updates for each layer. The `delta` for the current layer is equal to the delta of the previous layer, `D[-1]` dotted with the weight matrix of the current layer (**Line 109**). To finish off the computation of the `delta`, we multiply it by passing the activation for the `layer` through our derivative of the sigmoid (**Line 110**). We then update the deltas `D` list with the `delta` we just computed (**Line 111**).

Looking at this block of code we can see that the backpropagation step is iterative – we are simply taking the delta from the *previous layer*, dotting it with the weights of the *current layer*, and then multiplying by the derivative of the activation. This process is repeated until we reach the first layer in the network.

Given our deltas list `D`, we can move on to the weight update phase:

```
113
114
115         D = D[::- ]
116
117
118
119             layer    np.arange( ,    (    .W)):
120
121
122
123
124
125                 .W[layer] += -    .alpha * A[layer].T.dot(D[layer])
```

Keep in mind that during the backpropagation step we looped over our layers in *reverse* order. To perform our weight update phase, we'll simply *reverse* the ordering of entries in `D` so we can loop over each layer sequentially from 0 to N, the total number of layers in the network (**Line 115**).

Updating our actual weight matrix (i.e., where the actual "learning" takes place) is accomplished on **Line 125**, which is our gradient descent. We take the dot product of the current `layer` activation, `A[layer]` with the deltas of the current `layer`, `D[layer]` and multiple them by the learning rate, `alpha`. This value is added to the weight matrix for the current `layer`, `W[layer]`.

We repeat this process for all layers in the network. After performing the weight update phase, backpropagation is officially done.

Once our network is trained on a given dataset, we'll want to make predictions on the testing set, which can be accomplished via the `predict` method below:

```
127             (    , X, addBias=    ):
128
129
130
131         p = np.atleast_2d(X)
132
133
134            addBias:
135
136
137             p = np.c_[p, np.ones((p.shape[ ]))]
138
139
140             layer    np.arange( ,    (    .W)):
141
142
143
144
145
146                 p =     .sigmoid(np.dot(p,     .W[layer]))
147
148
149                 p
```

The `predict` function is simply a glorified forward pass. This function accepts one required parameter followed by a second optional one:

- `X`: The data points we'll be predicting class labels for.

- `addBias`: A boolean indicating whether we need to add a column of 1's to `X` to perform the bias trick.

On **Line 131** we initialize `p`, the output predictions as the input data points `X`. This value `p` will be passed through every layer in the network, propagating until we reach the final output prediction.

On **Lines 134-137** we make a check to see if the bias term should be embedded into the data points. If so, we insert a column of 1's as the last column in the matrix (exactly as we did in the `fit` method above).

From there, we perform the forward propagation by looping over all layers in our network on **Line 140**. The data points `p` are updated by taking the dot product between the current activations `p` and the weight matrix for the current `layer`, followed by passing the output through our sigmoid activation function (**Line 146**).

Given that we are looping over all layers in the network, we'll eventually reach the final layer, which will give us our final class label prediction. We return the predicted value to the calling function on **Line 149**.

The final function we'll define inside the `NeuralNetwork` class will be used to calculate the loss across our *entire* training set:

```
151                         (     , X, targets):
152
153
154             targets = np.atleast_2d(targets)
155             predictions =     .predict(X, addBias=     )
156             loss =     * np.sum((predictions - targets) **  )
157
158
159                     loss
```

The `calculate_loss` function requires that we pass in the data points `X` along with their ground-truth labels, `targets`. We make predictions on `X` on **Line 155** and then compute the sum squared error on **Line 156**. The loss is then returned to the calling function on **Line 159**. As our network learns, we should see this loss decrease.

Backpropagation with Python Example #1: Bitwise XOR

Now that we have implemented our `NeuralNetwork` class, let's go ahead and train it on the bitwise XOR dataset. As we know from our work with the Perceptron, this dataset is *not* linearly separable – our goal will be to train a neural network that can model this nonlinear function.

Go ahead and open up a new file, name it `nn_xor.py`, and insert the following code:

```
1
2       pyimagesearch.nn        NeuralNetwork
3         numpy    np
4
5
6   X = np.array([[ ,  ], [ ,  ], [ ,  ], [ ,  ]])
7   y = np.array([[ ], [ ], [ ], [ ]])
```

Lines 2 and 3 import our required Python packages. Notice how we are importing our newly implemented `NeuralNetwork` class. **Lines 6 and 7** then construct the XOR dataset, as depicted by Table 10.1 earlier in this chapter.

We can now define our network architecture and train it:

```
9
10 nn = NeuralNetwork([ ,  ,  ], alpha=   )
11 nn.fit(X, y, epochs=     )
```

On **Line 10** we instantiate our `NeuralNetwork` to have a $2-2-1$ architecture, implying there is:

1. An input layer with two nodes (i.e., our two inputs).
2. A single hidden layer with two nodes.
3. An output layer with one node.

Line 11 trains our network for a total of 20,000 epochs.

Once our network is trained, we'll loop over our XOR datasets, allow the network to predict the output for each one, and display the prediction to our screen:

```
13
14     (x, target)       (X, y):
15
16
17      pred = nn.predict(x)[ ][ ]
18      step =      pred >
19           ("[INFO] data={}, ground truth={}, pred={:.4f}, step={}".format(
20           x, target[ ], pred, step))
```

Line 18 applies a step function to the sigmoid output. If the prediction is > 0.5, we'll return *one*, otherwise, we will return *zero*. Applying this step function allows us to binarize our output class labels, just like the XOR function.

To train our neural network using backpropagation with Python, simply execute the following command:

```
python nn_xor.py
```

A plot of the squared loss is displayed below (Figure 10.11). As we can see, loss slowly decreases to approximately zero over the course of training. Furthermore, looking at the final four lines of the output we can see our predictions:

For each and every data point, our neural network was able to correctly learn the XOR pattern, demonstrating that our multi-layer neural network is capable of learning nonlinear functions.

To demonstrate that least one hidden layer is required to learn the XOR function, go back to **Line 10** where we define the $2-2-1$ architecture:

Figure 10.11: Loss over time for our $2-2-1$ neural network.

```
nn = NeuralNetwork([ ,  ,  ], alpha=   )
nn.fit(X, y, epochs=     )
```

And change it to be a 2-1 architecture:

```
define our  -  neural network     train it
nn = NeuralNetwork([ ,  ], alpha=   )
nn.fit(X, y, epochs=     )
```

From there, you can attempt to retrain your network:

```
  python nn_xor.py
```

No matter how much you fiddle with the learning rate or weight initializations, you'll never be able to approximate the XOR function. This fact is why multi-layer networks with nonlinear activation functions trained via backpropagation are so important – they enable us to learn patterns in datasets that are otherwise nonlinearly separable.

Backpropagation with Python Example: MNIST Sample

As a second, more interesting example, let's examine a subset of the MNIST dataset (Figure 10.12) for handwritten digit recognition. This subset of the MNIST dataset is built-into the scikit-learn library and includes 1,797 example digits, each of which are 8×8 grayscale images (the original images are 28×28. When flattened, these images are represented by an $8 \times 8 = 64$-dim vector.

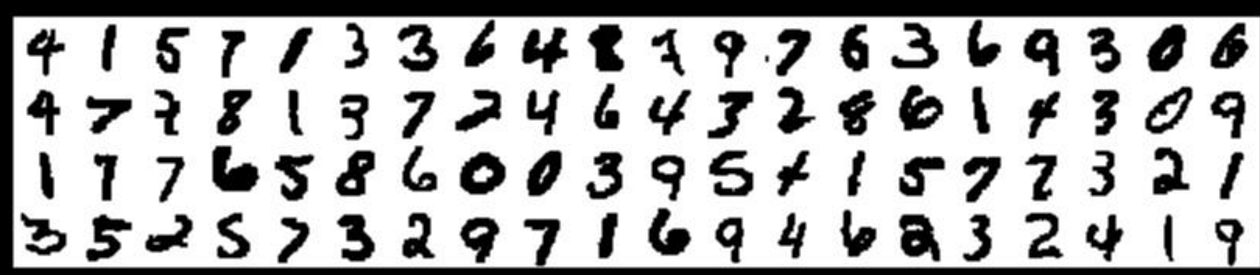

Figure 10.12: A sample of the MNIST dataset. The goal of this dataset is to correctly classify the handwritten digits, $0 - 9$.

Let's go ahead and train our `NeuralNetwork` implementation on this MNIST subset now. Open up a new file, name it `nn_mnist.py`, and we'll get to work:

```
1
2       pyimagesearch.nn        NeuralNetwork
3       sklearn.preprocessing        LabelBinarizer
4       sklearn.model_selection        train_test_split
5       sklearn.metrics        classification_report
6       sklearn        datasets
```

We start on **Lines 2-6** by importing our required Python packages.
From there, we load the MNIST dataset from disk using the scikit-learn helper functions:

```
8
9
10
11      ("[INFO] loading MNIST (sample) dataset...")
12  digits = datasets.load_digits()
13  data = digits.data.astype("float")
14  data = (data - data.min()) / (data.max() - data.min())
15      ("[INFO] samples: {}, dim: {}".format(data.shape[ ],
16      data.shape[ ]))
```

We also perform min/max normalizing by scaling each digit into the range $[0, 1]$ (**Line 14**).
Next, let's construct a training and testing split, using 75% of the data for testing and 25% for evaluation:

```
18
19  (trainX, testX, trainY, testY) = train_test_split(data,
20      digits.target, test_size=    )
21
22
23  trainY = LabelBinarizer().fit_transform(trainY)
24  testY = LabelBinarizer().fit_transform(testY)
```

We'll also encode our class label integers as vectors, a process called *one-hot encoding* that we will discuss in detail later in this chapter.

From there, we are ready to train our network:

```
26
27      ("[INFO] training network...")
28  nn = NeuralNetwork([trainX.shape[ ],   ,   ,   ])
29      ("[INFO] {}".format(nn))
30  nn.fit(trainX, trainY, epochs=    )
```

Here we can see that we are training a `NeuralNetwork` with a $64 - 32 - 16 - 10$ architecture. The output layer has ten nodes due to the fact that there are ten possible output classes for the digits *0-9*.

We then allow our network to train for 1,000 epochs. Once our network has been trained, we can evaluate it on the testing set:

```
32
33      ("[INFO] evaluating network...")
34  predictions = nn.predict(testX)
35  predictions = predictions.argmax(axis= )
36      (classification_report(testY.argmax(axis= ), predictions))
```

Line 34 computes the output `predictions` for every data point in `testX`. The `predictions` array has the shape `(450, 10)` as there are 450 data points in the testing set, each of which with ten possible class label probabilities.

To find the class label with the *largest probability* for *each data point*, we use the `argmax` function on **Line 35** – this function will return the index of the label with the highest predicted probability. We then display a nicely formatted classification report to our screen on **Line 36**.

To train our custom `NeuralNetwork` implementation on the MNIST dataset, just execute the following command:

```
python nn_mnist.py
```

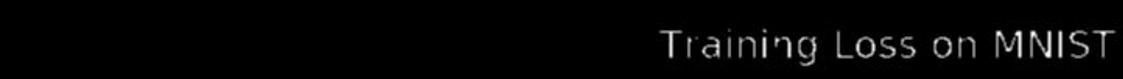

Figure 10.13: Plotting training loss on the MNIST dataset using a $64 - 32 - 16 - 10$ feedforward neural network.

I have included a plot of the squared loss as well (Figure 10.13). Notice how our loss starts off very high, but quickly drops during the training process. Our classification report demonstrates that we are obtaining $\approx 98\%$ classification accuracy on our testing set; however, we are having some trouble classifying digits 4 and 5 (95% and 94% accuracy, respectively). Later in this book, we'll learn how to train Convolutional Neural Networks on the *full* MNIST dataset and improve our accuracy further.

Backpropagation Summary

In this section, we learned how to implement the backpropagation algorithm from scratch using Python. Backpropagation is a generalization of the gradient descent family of algorithms that is specifically used to train multi-layer feedforward networks.

The backpropagation algorithm consists of two phases:

1. The forward pass where we pass our inputs through the network to obtain our output classifications.

2. The backward pass (i.e., weight update phase) where we compute the gradient of the loss function and use this information to iteratively apply the chain rule to update the weights in our network.

Regardless of whether we are working with simple feedforward neural networks or complex, deep Convolutional Neural Networks, the backpropagation algorithm is still used to train these models. This is accomplished by ensuring that the activation functions inside the network are *differentiable*, allowing the chain rule to be applied. Furthermore, any other layers inside the network that require updates to their weights/parameters, must also be compatible with backpropagation as well.

We implemented our backpropagation algorithm using the Python programming language and devised a multi-layer, feedforward `NeuralNetwork` class. This implementation was then trained on the XOR dataset to demonstrate that our neural network is capable of learning nonlinear functions by applying the backpropagation algorithm with at least one hidden layer. We then applied the same backpropagation + Python implementation to a subset of the MNIST dataset to demonstrate that the algorithm can be used to work with image data as well.

In practice, backpropagation can be not only challenging to implement (due to bugs in computing the gradient), but also hard to make efficient without special optimization libraries, which is why we often use libraries such as Keras, TensorFlow, and mxnet that have *already* (correctly) implemented backpropagation using optimized strategies.

Multi-layer Networks with Keras

Now that we have implemented neural networks in *pure Python*, let's move on to the preferred implementation method – using a dedicated (highly optimized) neural network library such as Keras.

In the next two sections, I'll discuss how to implement feedforward, multi-layer networks and apply them to the MNIST and CIFAR-10 datasets. These result will hardly be "state-of-the-art", but will serve two purposes:

- To demonstrate how you can implement simple neural networks using the Keras library.
- Obtain a baseline using standard neural networks which we will later compare to Convolutional Neural Networks (noting that CNNS will *dramatically* outperform our previous methods).

MNIST

Section 10.1.3 above, we only used a sample of the MNIST dataset for two reasons:

- To demonstrate how to implement your first feedforward neural network in pure Python.
- To facilitate faster result gathering – given that our pure Python implementation is by definition unoptimized, it will take longer to run.

Therefore, we used a *sample* of the dataset. In this section, we'll be using the *full* MNIST dataset, consisting of 70,000 data points (7,000 examples per digit). Each data point is represented by a 784-d vector, corresponding to the (flattened) 28×28 images in the MNIST dataset. Our goal is to train a neural network (using Keras) to obtain $> 90\%$ accuracy on this dataset.

As we'll find out, using Keras to build our network architecture is *substantially easier* than our pure Python version. In fact, the actual network architecture will only occupy *four lines of code* – the rest of the code in this example simply involves loading the data from disk, transforming the class labels, and then displaying the results.

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

```
1
2       sklearn.preprocessing        LabelBinarizer
```

```
3       sklearn.model_selection        train_test_split
4       sklearn.metrics        classification_report
5       keras.models        Sequential
6       keras.layers.core        Dense
7       keras.optimizers        SGD
8       sklearn        datasets
9         matplotlib.pyplot    plt
10        numpy    np
11        argparse
```

Lines 2-11 import our required Python packages. The `LabelBinarizer` will be used to one-hot encode our *integer labels* as *vector labels*. One-hot encoding transforms categorical labels from a single integer to a vector. Many machine learning algorithms (including neural networks) benefit from this type of label representation. I'll be discussing one-hot encoding in more detail and providing multiple examples (including using the `LabelBinarizer`) later in this section.

The `train_test_split` on **Line 3** will be used to create our training and testing splits from the MNIST dataset. The `classification_report` function will give us a nicely formatted report displaying the total accuracy of our model, along with a breakdown on the classification accuracy for *each* digit.

Lines 5-7 import the necessary packages to create a simple feedforward neural network with Keras. The `Sequential` class indicates that our network will be feedforward and layers will be added to the class *sequentially*, one on top of the other. The `Dense` class on **Line 6** is the implementation of our fully-connected layers. For our network to actually learn, we need to apply `SGD` (**Line 7**) to optimize the parameters of the network. Finally, to gain access to full MNIST dataset, we need to import the `datasets` helper from scikit-learn on **Line 8**.

Let's move on to parsing our command line arguments:

```
13
14  ap = argparse.ArgumentParser()
15  ap.add_argument(" o", "  output", required=    ,
16       help="path to the output loss/accuracy plot")
17  args =     (ap.parse_args())
```

We only need a single switch here, `--output`, which is the path to where our figure plotting the loss and accuracy over time will be saved to disk.

Next, let's load the full MNIST dataset:

```
19
20
21
22       ("[INFO] loading MNIST (full) dataset...")
23  dataset = datasets.fetch_mldata("MNIST Original")
24
25
26
27  data = dataset.data.astype("float") /
28  (trainX, testX, trainY, testY) = train_test_split(data,
29       dataset.target, test_size=    )
```

Line 23 loads the MNIST dataset from disk. If you have *never* run this function before, then the MNIST dataset will be downloaded and stored locally to your machine – this download is 55MB

and may take a minute or two to finish downloading, depending on your internet connection. Once the dataset has been downloaded, it is cached to your machine and will not have to be downloaded again.

We then perform data normalization on **Line 27** by scaling the pixel intensities to the range $[0, 1]$. We create a training and testing split, using 75% of the data for training and 25% for testing on **Lines 28 and 29**.

Given the training and testing splits, we can now encode our labels:

```
31
32 lb = LabelBinarizer()
33 trainY = lb.fit_transform(trainY)
34 testY = lb.transform(testY)
```

Each data point in the MNIST dataset has an integer label in the range $[0, 9]$, one for each of the possible ten digits in the MNIST dataset. A label with a value of 0 indicates that the corresponding image contains a zero digit. Similarly, a label with a value of 8 indicates that the corresponding image contains the number eight.

However, we first need to transform these *integer labels* into *vector labels*, where the index in the vector for label is set to 1 and 0 otherwise (this process is called *one-hot encoding*).

For example, consider the label 3 and we wish to binarize/one-hot encode it – the label 3 now becomes:

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

Notice how only the index for the digit three is set to one – all other entries in the vector are set to zero. Astute readers may wonder why the *fourth* and not the *third* entry in the vector is updated? Recall that the first entry in the label is actually for the digit zero. Therefore, the entry for the digit three is actually the fourth index in the list.

Here is a second example, this time with the label 1 binarized:

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

The second entry in the vector is set to one (since the first entry corresponds to the label 0), while all other entries are set to zero.

I have included the one-hot encoding representations for each digit, $0 - 9$, in the listing below:

```
0: [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
1: [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
2: [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
3: [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
4: [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
5: [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
6: [0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
7: [0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
8: [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
9: [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
```

This encoding may seem tedious, but many machine learning algorithms (including neural networks), benefit from this label representation. Luckily, most machine learning software packages provide a method/function to perform one-hot encoding, removing much of the tediousness.

Lines 32-34 simply perform this process of one-hot encoding the input *integer labels* as *vector labels* for both the training and testing set.

Next, let's define our network architecture:

```
36
37 model = Sequential()
38 model.add(Dense(   , input_shape=(   ,), activation="sigmoid"))
39 model.add(Dense(   , activation="sigmoid"))
40 model.add(Dense(  , activation="softmax"))
```

As you can see, our network is a feedforward architecture, instantiated by the `Sequential` class on **Line 37** – this architecture implies that the layers will be stacked on top of each other with the output of the previous layer feeding into the next.

Line 38 defines the first fully-connected layer in the network. The `input_shape` is set to 784, the dimensionality of each MNIST data points. We then learn 256 weights in this layer and apply the sigmoid activation function. The next layer (**Line 39**) learns 128 weights. Finally, **Line 40** applies another fully-connected layer, this time only learning 10 weights, corresponding to the ten (0-9) output classes. Instead of a sigmoid activation, we'll use a softmax activation to obtain normalized class probabilities for each prediction.

Let's go ahead and train our network:

```
42
43      ("[INFO] training network...")
44 sgd = SGD(    )
45 model.compile(loss="categorical_crossentropy", optimizer=sgd,
46      metrics=["accuracy"])
47 H = model.fit(trainX, trainY, validation_data=(testX, testY),
48      epochs=   , batch_size=   )
```

On **Line 44** we initialize the SGD optimizer with a learning rate of `0.01` (which we may commonly write as `1e-2`). We'll use the category cross-entropy loss function as our loss metric (**Lines 45 and 46**). Using the cross-entropy loss function is also why we had to convert our integer labels to vector labels.

A call to `.fit` of the `model` on **Line 47 and 48** kicks off the training of our neural network. We'll supply the training data and training labels as the first two arguments to the method.

The `validation_data` can then be supplied, which is our testing split. In *most* circumstances, such as when you are tuning hyperparameters or deciding on a model architecture, you'll want your validation set to be a *true* validation set and not your testing data. In this case, we are simply demonstrating how to train a neural network from scratch using Keras so we're being a bit lenient with our guidelines. Future chapters in this book, as well as the more advanced content in the *Practitioner Bundle* and *ImageNet Bundle*, are *much* more rigorous in the scientific method; however, for now, simply focus on the code and grasp how the network is trained.

We'll allow our network to train for a total of 100 epochs using a batch size of 128 data points at a time. The method returns a dictionary, `H`, which we'll use to plot the loss/accuracy of the network overtime in a couple of code blocks.

Once the network has finished training, we'll want to evaluate it on the testing data to obtain our final classifications:

```
50
51      ("[INFO] evaluating network...")
52  predictions = model.predict(testX, batch_size=   )
53      (classification_report(testY.argmax(axis= ),
54      predictions.argmax(axis= ),
55      target_names=[   (x)     x    lb.classes_]))
```

A call to the `.predict` method of `model` will return the class label probabilities for *every* data point in `testX` (**Line 52**). Thus, if you were to inspect the `predictions` NumPy array it would have the shape `(X, 10)` as there are 17,500 total data points in the testing set and ten possible class labels (the digits 0-9).

Each entry in a given row is, therefore, a *probability*. To determine the class with the *largest* probability, we can simply call `.argmax(axis=1)` as we do on **Line 53**, which will give us the *index* of the class label with the largest probability, and, therefore, our final output classification. The final output classification by the network is tabulated, and then a final classification report is displayed to our console on **Lines 53-55**.

Our final code block handles plotting the training loss, training accuracy, validation loss, and validation accuracy over time:

```
57
58  plt.style.use("ggplot")
59  plt.figure()
60  plt.plot(np.arange( ,    ), H.history["loss"], label="train_loss")
61  plt.plot(np.arange( ,    ), H.history["val_loss"], label="val_loss")
62  plt.plot(np.arange( ,    ), H.history["acc"], label="train_acc")
63  plt.plot(np.arange( ,    ), H.history["val_acc"], label="val_acc")
64  plt.title("Training Loss and Accuracy")
65  plt.xlabel("Epoch #")
66  plt.ylabel("Loss/Accuracy")
67  plt.legend()
68  plt.savefig(args["output"])
```

This plot is then saved to disk based on the `--output` command line argument.

To train our network of fully-connected layers on MNIST, just execute the following command:

```
python keras_mnist.py --output output/keras_mnist.png
```

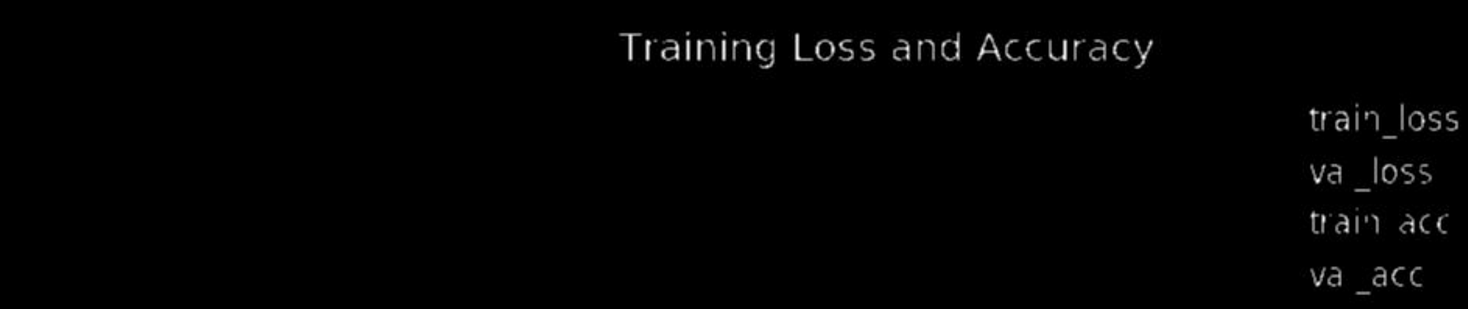

Figure 10.14: Training a $784 - 256 - 128 - 10$ feedforward neural network with Keras on the *full* MNIST dataset. Notice how our training and validation curves are near identical, implying there is no overfitting occurring.

As the results demonstrate, we are obtaining $\approx 92\%$ accuracy. Furthermore, the training and validation curves match each other *nearly identically* (Figure 10.14), indicating there is no overfitting or issues with the training process.

In fact, if you are unfamiliar with the MNIST dataset, you might think 92% accuracy is *excellent* – and it was, perhaps 20 years ago. As we'll find out in Chapter 14, using Convolutional Neural Networks, we can easily obtain $> 98\%$ accuracy. Current state-of-the-art approaches can even break 99% accuracy.

1. Introduction

"The secret of getting ahead is to get started." – Mark Twain

Welcome to *Deep Learning for Computer Vision with Python*. This book is your guide to mastering deep learning applied to practical, real-world computer vision problems utilizing the Python programming language and the Keras + mxnet libraries. Inside this book, you'll learn how to apply deep learning to take-on projects such as image classification, object detection, training networks on large-scale datasets, and *much more*.

Deep Learning for Computer Vision with Python strives to be the *perfect balance* between ***theory taught in a classroom/textbook and the actual hands-on knowledge you'll need to be successful in the real world***.

To accomplish this goal, you'll learn in a *practical, applied* manner by training networks on your own custom datasets and even competing in challenging state-of-the-art image classification challenges and competitions. By the time you finish this book, you'll be *well equipped* to apply deep learning to your own projects. And with enough practice, I have no doubt that you'll be able to leverage your newly gained knowledge to find a job in the deep learning space, become a deep learning for computer vision consultant/contractor, or even start your own computer vision-based company that leverages deep learning.

So grab your highlighter. Find a comfortable spot. And let me help you on your journey to deep learning mastery. Remember the most important step is the first one – to simply get started.

I Studied Deep Learning the *Wrong Way*...This Is the *Right Way*

I want to start this book by sharing a personal story with you:

Toward the end of my graduate school career (2013-2014), I started wrapping my head around this whole "deep learning" thing due to a timing quirk. I was in a very unique situation. My dissertation was (essentially) wrapped up. Each of my Ph.D. committee members had signed off on it. However, due to university/department regulations, I still had an extra semester that I needed to "hang around" for before I could officially defend my dissertation and graduate. This gap essentially

While on the surface it may appear that our (strictly) fully-connected network is performing well, we can actually do much better. And as we'll see in the next section, strictly fully-connected networks applied to more challenging datasets can in some cases do just barely better than guessing randomly.

CIFAR-10

When it comes to computer vision and machine learning, the MNIST dataset is the classic definition of a "benchmark" dataset, one that is too easy to obtain high accuracy results on, and not representative of the images we'll see in the real world.

For a more challenging benchmark dataset, we commonly use CIFAR-10, a collection of 60,000, 32×32 RGB images, thus implying that each image in the dataset is represented by $32 \times 32 \times 3 = 3,072$ integers. As the name suggests, CIFAR-10 consists of 10 classes, including *airplane*, *automobile*, *bird*, *cat*, *deer*, *dog*, *frog*, *horse*, *ship*, and *truck*. A sample of the CIFAR-10 dataset for each class can be seen in Figure 10.15.

Figure 10.15: Example images from the ten class CIFAR-10 dataset.

Each class is evenly represented with 6,000 images per class. When training and evaluating a machine learning model on CIFAR-10, it's typical to use the predefined data splits by the authors and use 50,000 images for training and 10,000 for testing.

CIFAR-10 is *substantially harder* than the MNIST dataset. The challenge comes from the dramatic variance in how objects appear. For example, we can no longer assume that an image containing a green pixel at a given (x, y)-coordinate is a frog. This pixel could be a background of a forest that contains a deer. Or it could be the color of a green car or truck.

These assumptions are a stark contrast to the MNIST dataset, where the network can learn assumptions regarding the spatial distribution of pixel intensities. For example, the spatial distribution of foreground pixels of a 1 is substantially different than a 0 or a 5. This type of variance in object appearance makes applying a series of fully-connected layers much more challenging. As we'll find out in the rest of this section, standard FC (fully-connected) layer networks are not suited for this type of image classification.

Let's go ahead and get started. Open up a new file, name it `keras_cifar10.py`, and insert the following code:

```
1
2       sklearn.preprocessing        LabelBinarizer
3       sklearn.metrics        classification_report
4       keras.models        Sequential
5       keras.layers.core        Dense
6       keras.optimizers        SGD
7       keras.datasets        cifar10
8         matplotlib.pyplot    plt
9         numpy    np
10        argparse
```

Lines 2-10 import our required Python packages to build our fully-connected network, identical to the previous section with MNIST. The exception is the special utility function on **Line 7** – since CIFAR-10 is such a common dataset that researchers benchmark machine learning and deep learning algorithms on, it's common to see deep learning libraries provide simple helper functions to *automatically* load this dataset from disk.

Next, we can parse our command line arguments:

```
16
17  ap = argparse.ArgumentParser()
18  ap.add_argument(" o", "  output", required=    ,
19       help="path to the output loss/accuracy plot")
20  args =     (ap.parse_args())
```

The only command line argument we need is `--output`, the path to our output loss/accuracy plot.

Let's go ahead and load the CIFAR-10 dataset:

```
18
19
20       ("[INFO] loading CIFAR 10 data...")
21  ((trainX, trainY), (testX, testY)) = cifar10.load_data()
22  trainX = trainX.astype("float") /
23  testX = testX.astype("float") /
24  trainX = trainX.reshape((trainX.shape[ ],     ))
25  testX = testX.reshape((testX.shape[ ],     ))
```

A call to `cifar10.load_data` on **Line 21** automatically loads the CIFAR-10 dataset from disk, pre-segmented into training and testing split. If this is the *first* time you are calling `cifar10.load_data`, then this function will fetch and download the dataset for you. This file is $\approx 170MB$, so be patient as it is downloaded and unarchived. Once the file is downloaded once, it will be cached locally on your machine and will not have to be downloaded again.

Lines 22 and 23 convert the data type CIFAR-10 from unsigned 8-bit integers to floating point, followed by scaling the data to the range $[0, 1]$. **Lines 24 and 25** are responsible for *reshaping* the design matrix for the training and testing data. Recall that each image in the CIFAR-10 dataset is represented by a $32 \times 32 \times 3$ image.

For example, `trainX` has the shape `(50000, 32, 32, 3)` and `testX` has the shape `(10000, 32, 32, 3)`. If we were to *flatten* this image into a single list of floating point values, the list would have a total of $32 \times 32 \times 3 = 3,072$ total entries in it.

To flatten each of the images in the training and testing sets, we simply use the `.reshape` function of NumPy. After this function executes, `trainX` now has the shape `(50000, 3072)` while `testX` has the shape `(10000, 3072)`.

Now that the CIFAR-10 dataset has been loaded from disk, let's once again binarize the class label integers into vectors, followed by initializing a list of the actual *names* of the class labels:

```
27
28  lb = LabelBinarizer()
29  trainY = lb.fit_transform(trainY)
30  testY = lb.transform(testY)
31
32
33  labelNames = ["airplane", "automobile", "bird", "cat", "deer",
34       "dog", "frog", "horse", "ship", "truck"]
```

It's now time to define the network architecture:

```
36
37  model = Sequential()
38  model.add(Dense(    , input_shape=(    ,), activation="relu"))
39  model.add(Dense(   , activation="relu"))
40  model.add(Dense(  , activation="softmax"))
```

Line 37 instantiates the `Sequential` class. We then add the first `Dense` layer which has an `input_shape` of 3072, a node for each of the 3,072 flattened pixel values in the design matrix – this layer is then responsible for learning 1,024 weights. We'll also swap out the antiquated sigmoid for a ReLU activation in hopes of improve network performance.

The next fully-connected layer (**Line 39**) learns 512 weights, while the final layer (**Line 40**) learns weights corresponding to ten possible output classifications, along with a softmax classifier to obtain the final output probabilities for each class.

Now that the architecture of the network is defined, we can train it:

```
42
43       ("[INFO] training network...")
44  sgd = SGD(    )
45  model.compile(loss="categorical_crossentropy", optimizer=sgd,
46       metrics=["accuracy"])
47  H = model.fit(trainX, trainY, validation_data=(testX, testY),
48       epochs=   , batch_size=  )
```

We'll use the SGD optimizer to train the network with a learning rate of `0.01`, a fairly standard initial choice. The network will be trained for a total of 100 epochs using batches of 32.

Once the network has been trained, we can evaluate it using `classification_report` to obtain a more detailed review of model performance:

```
50
51       ("[INFO] evaluating network...")
52  predictions = model.predict(testX, batch_size=  )
53       (classification_report(testY.argmax(axis= ),
54       predictions.argmax(axis= ), target_names=labelNames))
```

And finally, we'll also plot the loss/accuracy over time as well:

```

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

To train our network on CIFAR-10, open up a terminal and execute the following command:

```
python keras_cifar10.py --output output/keras_cifar10.png
```

Looking at the output, you can see that our network obtained 57% accuracy. Examining our plot of loss and accuracy over time (Figure 10.16), we can see that our network struggles with overfitting past epoch 10. Loss initially starts to decrease, levels out a bit, and then skyrockets, and never comes down again. All the while training loss is falling consistently epoch-over-epoch.

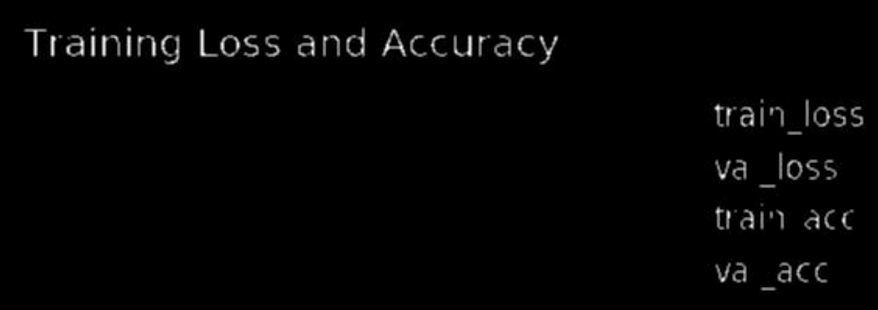

Figure 10.16: Using a standard feedforward neural network leads to dramatic overfitting in the more challenging CIFAR-10 dataset (notice how training loss falls while validation loss rises dramatically). To be successful at the CIFAR-10 challenge, we'll need a powerful technique – Convolutional Neural Networks.

This behavior of *decreasing* training loss while validation loss *increases* is indicative of *extreme overfitting*.

We could certainly consider optimizing our hyperparameters further, in particular, experimenting with varying learning rates and increasing both the depth and the number of nodes in the network, but we would be fighting for meager gains.

The fact is this – basic feedforward networks with strictly fully-connected layers are not suitable for challenging image datasets. For that, we need a more advanced approach: Convolutional Neural Networks. Luckily, CNNs are the topic of the entire remainder of this book. By the time you finish the *Starter Bundle*, you'll be able to obtain over 79% accuracy on CIFAR-10. If you so choose to study deep learning in more depth, the *Practitioner Bundle* will demonstrate how to increase your accuracy to over 93%, putting us in the league of state-of-the-art results [110].

The Four Ingredients in a Neural Network Recipe

You might have started to notice a pattern in our Python code examples when training neural networks. There are four main ingredients you need to put together your own neural network and deep learning algorithm: a *dataset*, a *model/architecture*, a *loss function*, and an *optimization method*. We'll review each of these ingredients below.

Dataset

The dataset is the first ingredient in training a neural network – the data itself along with the problem we are trying to solve define our end goals. For example, are we using neural networks to perform a regression analysis to predict the value of homes in a specific suburb in 20 years? Is our goal

to perform unsupervised learning, such as dimensionality reduction? Or are we trying to perform classification?

In the context of this book, we're strictly on *image classification*; however, the combination of your dataset and the problem you are trying to solve influences your choice in loss function, network architecture, and optimization method used to train the model. Usually, we have little choice in our dataset (unless you're working on a hobby project) – we are given a dataset with some expectation on what the results from our project should be. It is then up to us to train a machine learning model on the dataset to perform well on the given task.

Loss Function

Given our dataset and target goal, we need to define a loss function that aligns with the problem we are trying to solve. In nearly all image classification problems using deep learning, we'll be using cross-entropy loss. For > 2 classes we call this *categorical cross-entropy*. For two class problems, we call the loss *binary cross-entropy*.

Model/Architecture

Your network architecture can be considered the first actual "choice" you have to make as an ingredient. Your dataset is likely chosen for you (or at least you've *decided* that you want to work with a given dataset). And if you're performing classification, you'll in all likelihood be using cross-entropy as your loss function.

However, your network architecture can vary dramatically, especially when with which optimization method you choose to train your network. After taking the time to explore your dataset and look at:

1. How many data points you have.
2. The number of classes.
3. How similar/dissimilar the classes are.
4. The intra-class variance.

You should start to develop a "feel" for a network architecture you are going to use. This takes practice as deep learning is part science, part art – in fact, the rest of this book is dedicated to helping you develop both of these skills.

Keep in mind that the number of layers and nodes in your network architecture (along with any type of regularization) is likely to change as you perform more and more experiments. The more results you gather, the better equipped you are to make informed decisions on which techniques to try next.

Optimization Method

The final ingredient is to define an optimization method. As we've seen thus far in this book *Stochastic Gradient Descent* (Section 9.2) is used quite often. Other optimization methods exist, including RMSprop [90], Adagrad [111], Adadelta [112], and Adam [113]; however, these are more advanced optimization methods that we'll cover in the *Practitioner Bundle*.

Even despite all these newer optimization methods, SGD is *still* the workhorse of deep learning – most neural networks are trained via SGD, including the networks obtaining state-of-the-art accuracy on challenging image datasets such as ImageNet.

When training deep learning networks, especially when you're first getting started and learning the ropes, *SGD should be your optimizer of choice*. You then need to set a proper learning rate and regularization strength, the total number of epochs the network should be trained for, and whether or not momentum (and if so, which value) or Nesterov acceleration should be used. Take the time to experiment with SGD *as much as you possibly can* and *become comfortable with tuning the parameters*.

Becoming familiar with a given optimization algorithm is similar to mastering how to drive a

car – you drive your own car better than other people's cars because you've spent so much time driving it; you understand your car and its intricacies. Often times, a given optimizer is chosen to train a network on a dataset *not* because the optimizer itself is better, but because the *driver* (i.e., deep learning practitioner) is *more familiar with the optimizer* and understands the "art" behind tuning its respective parameters.

Keep in mind that obtaining a reasonably performing neural network on even a small/medium dataset can take 10's to 100's of experiments even for advanced deep learning users – don't be discouraged when your network isn't performing extremely well right out of the gate. Becoming proficient in deep learning will require an investment of your time and *many* experiments – but it will be worth it once you master how these ingredients come together.

Weight Initialization

Before we close out this chapter I wanted to briefly discuss the concept of *weight initialization*, or more simply, how we initialize our weight matrices and bias vectors.

This section is not meant to be a comprehensive initialization techniques; however, it does highlight popular methods, but from neural network literature and general rules-of-thumb. To illustrate how these weight initialization methods work I have included basic Python/NumPy-like pseudocode when appropriate.

Constant Initialization

When applying constant normalization, all weights in the neural network are initialized with a constant value, C. Typically C will equal zero or one.

To visualize this in pseudocode let's consider an arbitrary layer of a neural network that has 64 inputs and 32 outputs (excluding any biases for notional convenience). To initialize these weights via NumPy and zero initialization (the default used by Caffe, a popular deep learning framework) we would execute:

```
>>> W = np.zeros((  ,   ))
```

Similarly, one initialization can be accomplished via:

```
>>> W = np.ones((  ,   ))
```

We can apply constant initialization using an arbitrary of C using:

```
>>> W = np.ones((  ,   )) * C
```

Although constant initialization is easy to grasp and understand, the problem with using this method is that its near impossible for us to break the symmetry of activations [114]. Therefore, it is rarely used as a neural network weight initializer.

Uniform and Normal Distributions

A *uniform distribution* draws a random value from the range `[lower, upper]` where every value inside this range has *equal probability* of being drawn.

Again, let's presume that for a given layer in a neural network we have 64 inputs and 32 outputs. We then wish to initialize our weights in the range `lower=-0.05` and `upper=0.05`. Applying the following Python + NumPy code will allow us to achieve the desired normalization:

```
>>> W = np.random.uniform(low=-    , high=    , size=(  ,   ))
```

Executing the code above NumPy will randomly generate $64 \times 32 = 2,048$ values from the range $[-0.05, 0.05]$ where each value in this range has equal probability.

We then have a *normal distribution* where we define the probability density for the Gaussian distribution as:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{10.6}$$

The most important parameters here are μ (the mean) and σ (the standard deviation). The square of the standard deviation, σ^2, is called the variance.

When using the Keras library the `RandomNormal` class draws random values from a normal distribution with $\mu = 0$ and $\sigma = 0.05$. We can mimic this behavior using NumPy below:

```
>>> W = np.random.normal(   ,    , size=(  ,   ))
```

Both of uniform and normal distributions can be used to initialize the weights in neural networks; however, we normally impose various heuristics to create "better" initialization schemes (as we'll discuss in the remaining sections).

LeCun Uniform and Normal

If you have ever used the Torch7 or PyTorch frameworks you may notice that the default weight initialization method is called "Efficient Backprop", which is derived by the work of LeCun et al. [17].

Here the authors define a parameter F_{in} (called "fan in", or the number of *inputs* to the layer) along with F_{out} (the "fan out", or number of *outputs* from the layer). Using these values we can apply uniform initialization by:

```
>>> F_in = 64
>>> F_out = 32
>>> limit = np.sqrt(3 / float(F_in))
>>> W = np.random.uniform(low=-limit, high=limit, size=(F_in, F_out))
```

We can also use a normal distribution as well. The Keras library uses a truncated normal distribution when constructing the lower and upper limits, along with a zero mean:

```
>>> F_in = 64
>>> F_out = 32
>>> limit = np.sqrt(1 / float(F_in))
>>> W = np.random.normal(0.0, limit, size=(F_in, F_out))
```

Glorot/Xavier Uniform and Normal

The default weight initialization method used in the Keras library is called "Glorot initialization" or "Xavier initialization" named after Xavier Glorot, the first author of the paper, *Understanding the difficulty of training deep feedforward neural networks* [115].

For the normal distribution the `limit` value is constructed by *averaging* the F_{in} and F_{out} together and then taking the square-root [116]. A zero-center ($\mu = 0$) is then used:

```
>>> F_in = 64
>>> F_out = 32
>>> limit = np.sqrt(2 / float(F_in + F_out))
>>> W = np.random.normal(0.0, limit, size=(F_in, F_out))
```

Glorot/Xavier initialization can also be done with a uniform distribution where we place stronger restrictions on `limit`:

```
>>> F_in = 64
>>> F_out = 32
>>> limit = np.sqrt(6 / float(F_in + F_out))
>>> W = np.random.uniform(low=-limit, high=limit, size=(F_in, F_out))
```

Learning tends to be quite efficient using this initialization method and I recommend it for most neural networks.

He et al./Kaiming/MSRA Uniform and Normal

Often referred to as "He et al. initialization", "Kaiming initialization", or simply "MSRA initialization", this technique is named after Kaiming He, the first author of the paper, *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification* [117].

We typically used this method when we are training *very deep* neural networks that use a ReLU-like activation function (in particular, a "PReLU", or Parametric Rectified Linear Unit).

To initialize the weights in a layer using He et al. initialization with a *uniform distribution* we set `limit` to be $limit = \sqrt{6/F_{in}}$, where F_{in} is the number of input units in the layer:

```
>>> F_in = 64
>>> F_out = 32
>>> limit = np.sqrt(6 / float(F_in))
>>> W = np.random.uniform(low=-limit, high=limit, size=(F_in, F_out))
```

We can also use a *normal distribution* as well by setting $\mu = 0$ and $sigma = \sqrt{2/F_{in}}$

```
>>> F_in = 64
>>> F_out = 32
>>> limit = np.sqrt(2 / float(F_in))
>>> W = np.random.normal(0.0, limit, size=(F_in, F_out))
```

We'll discuss this initialization method in both the *Practitioner Bundle* and the *ImageNet Bundle* of this book where we train very deep neural networks on large image datasets.

Differences in Initialization Implementation

The actual `limit` values may vary for LeCun Uniform/Normal, Xavier Uniform/Normal, and He et al. Uniform/Normal. For example, when using Xavier Uniform in Caffe, `limit = -np.sqrt(3 / F_in)` [114]; however, the default Xaiver initialization for Keras uses `np.sqrt(6 / (F_in + F_out))` [118]. No method is "more correct" than the other, but you should read the documentation of your respective deep learning library.

Summary

In this chapter, we reviewed the fundamentals of neural networks. Specifically, we focused on the history of neural networks and the relation to biology.

From there, we moved on to *artificial neural networks*, such as the Perceptron algorithm. While important from a historical standpoint, the Perceptron algorithm has one major flaw – it accurately classifies nonlinear separable points. In order to work with more challenging datasets we need both (1) nonlinear activation functions and (2) multi-layer networks.

To train multi-layer networks we must use the backpropagation algorithm. We then implemented backpropagation by hand and demonstrated that when used to train multi-layer networks with nonlinear activation functions, we can model nonlinearly separable datasets, such as XOR.

Of course, implementing backpropagation by hand is an arduous process prone to bugs – we, therefore, often rely on existing libraries such as Keras, Theano, TensorFlow, etc. This enables us to focus on the actual *architecture* rather than the underlying algorithm used to train the network.

Finally, we reviewed the four key ingredients when working with *any* neural network, including the *dataset*, *loss function*, *model/architecture*, and *optimization method*.

Unfortunately, as some of our results demonstrated (e.x., CIFAR-10) standard neural networks fail to obtain high classification accuracy when working with challenging image datasets that exhibit variations in translation, rotation, viewpoint, etc. In order to obtain reasonable accuracy on these datasets, we'll need to work with a special type of feedforward neural networks called *Convolutional Neural Networks* (CNNs), which is exactly the subject of our next chapter.

left me with an entire semester ($\approx$ 4 months) to kill – **it was an *excellent* time to start studying deep learning**.

My first stop, as is true for most academics, was to read through all the recent publications on deep learning. Due to my machine learning background, it didn't take long to grasp the actual *theoretical foundations* of deep learning.

However, I'm of the opinion that until you take your *theoretical knowledge* and *implement it*, you haven't actually *learned* anything yet. Transforming *theory* into *implementation* is a **very** different process, as any computer scientist who has taken a data structures class before will tell you: *reading about* red-black trees and then actually *implementing them from scratch* requires two different skill sets.

And that's *exactly* what my problem was.

After reading these deep learning publications, I was left scratching my head; I couldn't take what I learned from the papers and *implement* the actual algorithms, let alone *reproduce* the results.

Frustrated with my failed attempts at implementation, I spent *hours* searching on Google, hunting for deep learning tutorials, only to come up empty-handed. Back then, there weren't many deep learning tutorials to be found.

Finally, I resorted to playing around with libraries and tools such as Caffe, Theano, and Torch, blindly followed poorly written blog posts (with mixed results, to say the least).

I *wanted* to get started, but nothing had actually *clicked* yet – the deep learning lightbulb in my head was stuck in the "off" position.

To be totally honest with you, it was a painful, emotionally trying semester. I could *clearly* see the value of deep learning for computer vision, but I had nothing to show for my effort, except for a stack of deep learning papers on my desk that I *understood* but struggled to *implement*.

During the last month of the semester, I finally found my way to deep learning success through hundreds of trial-and-error experiments, countless late nights, and *a lot* of perseverance. In the long run, those four months made a massive impact on my life, my research path, and how I understand deep learning today...

...but I would *not* advise you to take the same path I did.

If you take *anything* from my personal experience, it should be this:

1. You don't need a decade of theory to get started in deep learning.
2. You don't need pages and pages of equations.
3. And you certainly don't need a degree in computer science (although it can be helpful).

When I got started studying deep learning, I made the critical mistake of taking a deep dive into the publications without ever resurfacing to try and implement what I studied. Don't get me wrong – theory *is* important. But if you don't (or can't) take your newly minted theoretical knowledge and use apply it to build actual real-world applications, you'll struggle to find your space in the deep learning world.

Deep learning, and most other higher-level, specialized computer science subjects are recognizing that *theoretical knowledge is not enough* – **we need to be *practitioners* in our respective fields as well**. In fact, the concept of becoming a deep learning *practitioner* was my ***exact motivation*** in writing *Deep Learning for Computer Vision with Python*.

While there are:

1. Textbooks that will teach you the theoretical underpinnings of machine learning, neural networks, and deep learning
2. And countless "cookbook"-style resources that will "show you in code", but never relate the code back to true theoretical knowledge...

...*none* of these books or resources will serve as the bridge between the other.

On one side of the bridge you have your textbooks, deeply rooted in theory and abstraction. And on the other side, you have "show me in code" books that simply present examples to you,

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.

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.

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].

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.

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***.

> 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.

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.

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
2       skimage.exposure        rescale_intensity
3         numpy     np
4         argparse
5         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               (image, K):
8
9       (iH, iW) = image.shape[: ]
10      (kH, kW) = K.shape[: ]
11
12
13
14
15      pad = (kW -  ) //
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
21
```

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          y    np.arange(pad, iH + pad):
23               x    np.arange(pad, iW + pad):
24
25
26
27                roi = image[y - pad:y + pad +  , x - pad:x + pad +  ]
28
29
30
31
32                k = (roi * K).sum()
33
34
35
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
39      output = rescale_intensity(output, in_range=( ,    ))
40      output = (output *    ).astype("uint8")
41
42
43              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
46  ap = argparse.ArgumentParser()
47  ap.add_argument(" i", "  image", required=    ,
48      help="path to the input image")
49  args =     (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
52  smallBlur = np.ones(( ,  ), dtype="float") * (    / (  *  ))
53  largeBlur = np.ones((  ,   ), dtype="float") * (    / (   *   ))
```

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
56  sharpen = np.array((
57      [ , - ,  ],
58      [- ,  , - ],
59      [ , - ,  ]), dtype="int")
```

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

```


laplacian = np.array((
    [ ,  ,  ],
    [ , - ,  ],
    [ ,  ,  ]), dtype="int")
```

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

```

sobelX = np.array((
    [- ,  ,  ],
    [- ,  ,  ],
    [- ,  ,  ]), dtype="int")


sobelY = np.array((
    [- , - , - ],
    [ ,  ,  ],
    [ ,  ,  ]), dtype="int")
```

And finally, we define the emboss kernel:

```

emboss = np.array((
    [- , - ,  ],
    [- ,  ,  ],
    [ ,  ,  ]), 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":

```


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
99  image = cv2.imread(args["image"])
100 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
101
102
103     (kernelName, K)    kernelBank:
104
105
106         ("[INFO] applying {} kernel".format(kernelName))
107     convolveOutput = convolve(gray, K)
108     opencvOutput = cv2.filter2D(gray, - , K)
109
110
111     cv2.imshow("Original", gray)
112     cv2.imshow("{}   convole".format(kernelName), convolveOutput)
113     cv2.imshow("{}   opencv".format(kernelName), opencvOutput)
114     cv2.waitKey( )
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*).

perhaps explaining the code, but never relating the code back to the underlying theory.

There is a fundamental disconnect between these two styles of learning, a gap that I want to help fill so you can learn in a *better, more efficient* way.

I thought back to my graduate school days, to my feelings of frustration and irritation, to the days when I even considered giving up. I channeled these feelings as I sat down to write this book. ***The book you're reading now is the book I wish I had when I first started studying deep learning.***

Inside the remainder of *Deep Learning for Computer Vision with Python*, you'll find **super practical walkthroughs**, **hands-on tutorials (with lots of code)**, and a **no-nonsense teaching style** that is guaranteed to cut through all the cruft and help you master deep learning for computer vision.

Rest assured, you're in good hands – this is the *exact* book that you've been looking for, and I'm incredibly excited to be joining you on your deep learning for visual recognition journey.

Who This Book Is For

This book is for **developers**, **researchers**, and **students** who want to become proficient in deep learning for computer vision and visual recognition.

Just Getting Started in Deep Learning?

Don't worry. You won't get bogged down by tons of theory and complex equations. We'll start off with the basics of machine learning and neural networks. You'll learn in a fun, practical way with lots of code. I'll also provide you with references to seminal papers in the machine learning literature that you can use to extend your knowledge once you feel like you have a solid foundation to stand on.

The most important step you can take right now is to simply *get started*. Let me take care of the teaching – regardless of your skill level, trust me, you will not get left behind. By the time you finish the first few chapters of this book, you'll be a neural network ninja and be able to graduate to the more advanced content.

Already a Seasoned Deep Learning Practitioner?

This book isn't just for beginners – *there's advanced content in here too*. For each chapter in this book, I provide a set of academic references you can use to further your knowledge. Many chapters inside *Deep Learning for Computer Vision with Python* actually *explain* these academic concepts in a manner that is easily understood and digested.

Best of all, the solutions and tactics I provide inside this book can be directly applied to your current job and research. The time you'll save by reading through *Deep Learning for Computer Vision with Python* will *more than pay for itself* once you apply your knowledge to your projects/research.

Book Organization

Since this book covers a *huge* amount of content, I've broken down the book into **three volumes** called ***"bundles"***. Each bundle sequentially builds on top of the other and includes all chapters from the lower volumes. You can find a quick breakdown of the bundles below.

Volume #1: Starter Bundle

The Starter Bundle is a great fit if you're taking your first steps toward deep learning for image classification mastery.

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*.

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.

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.

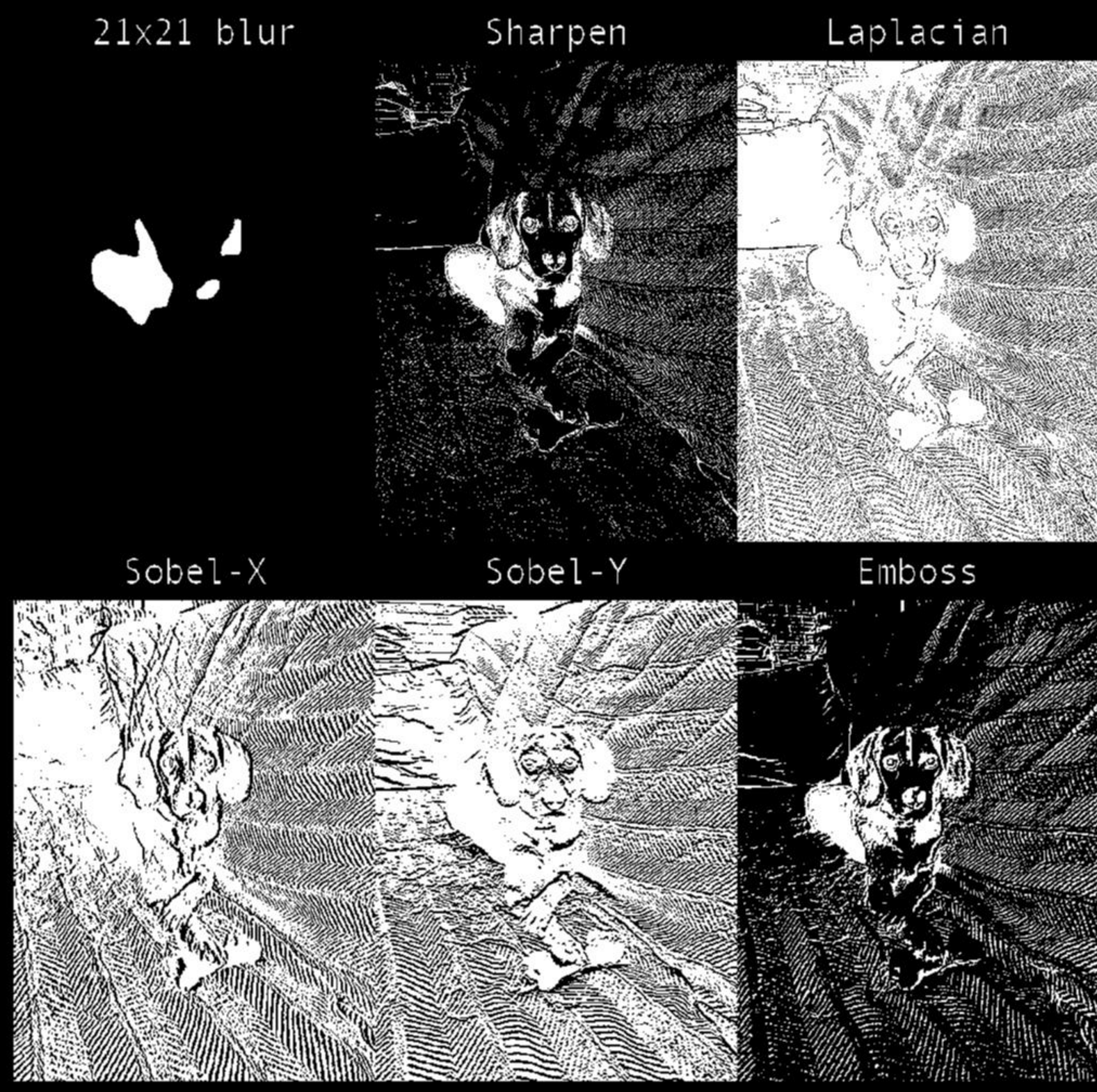

Figure 11.5: **Top-left:** Applying a 21×21 average blur. Notice how this image is more blurred than in Figure 11.4. **Top-mid:** Using a sharpening kernel to enhance details. **Top-right:** Detecting edge-like operations via the Laplacian operator. **Bottom-left:** Computing vertical edges using the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

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.

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.

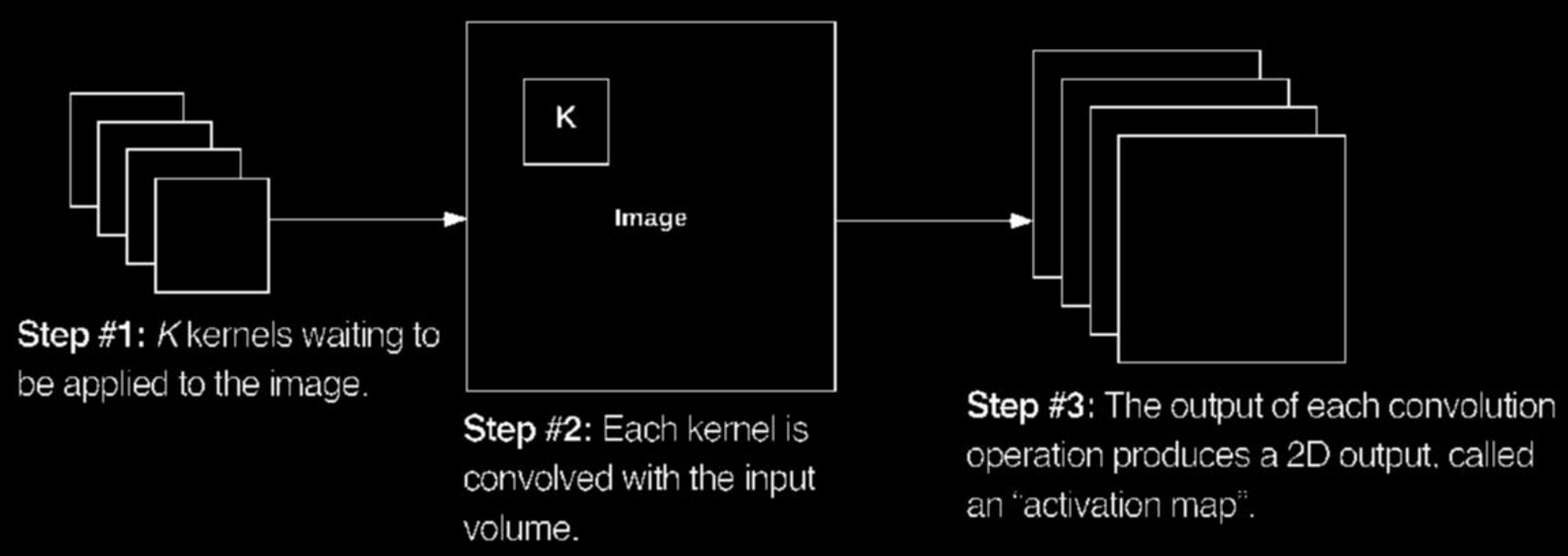

Figure 11.6: **Left:** At each convolutional layer in a CNN, there are K kernels applied to the input volume. **Middle:** Each of the K kernels is convolved with the input volume. **Right:** Each kernel produces an 2D output, called an *activation map*.

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).

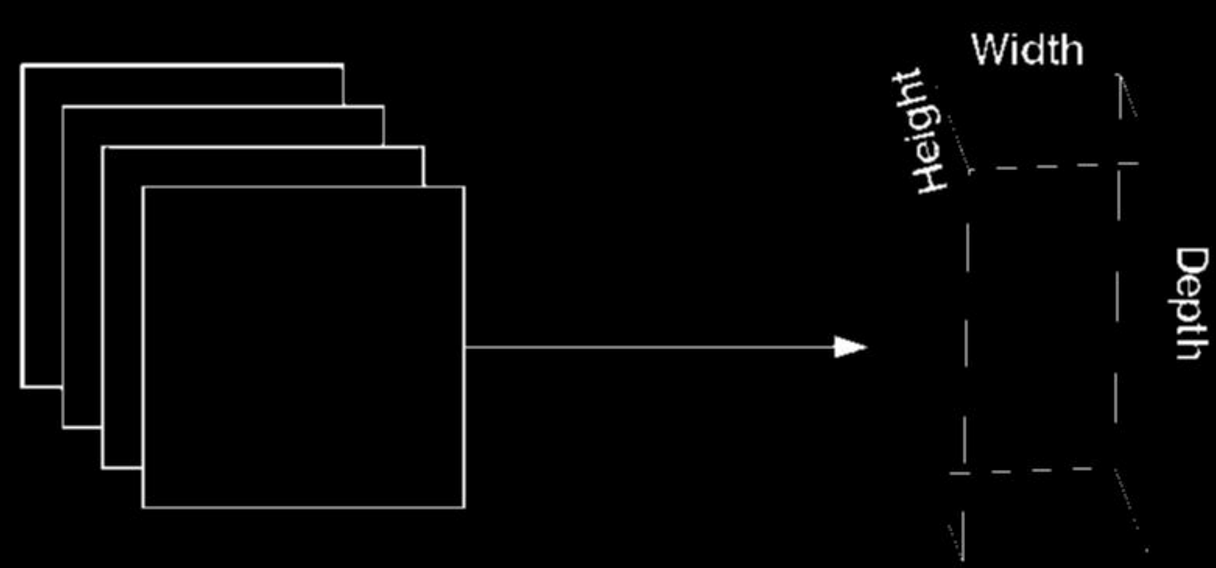

Figure 11.7: After obtaining the K activation maps, they are stacked together to form the input volume to the next layer in the network.

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* ( ) 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.

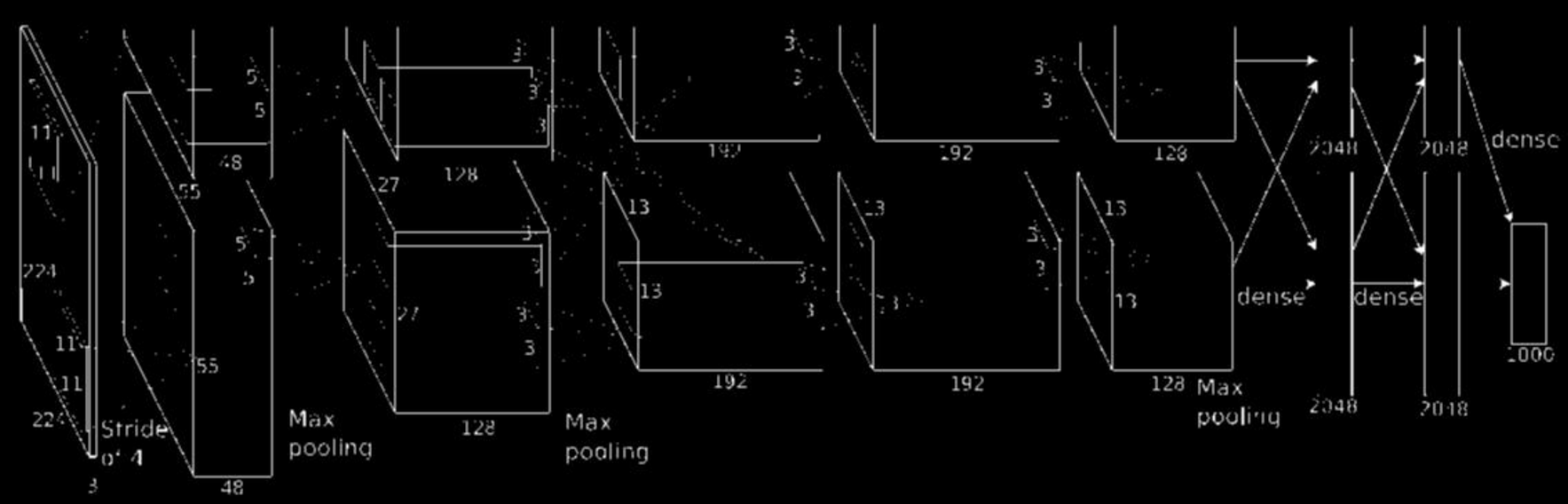

Figure 11.8: The original AlexNet architecture diagram provided by Krizhevsky et al. [94] Notice how the input image is documented to be $224 \times 224 \times 3$, although this cannot be possible due to Equation 11.6. It's also worth noting that we are unsure why the top-half of this figure is cutoff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

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, $W_{input} = W_{output}$, $H_{input} = H_{output}$, $D_{input} = D_{output}$.

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

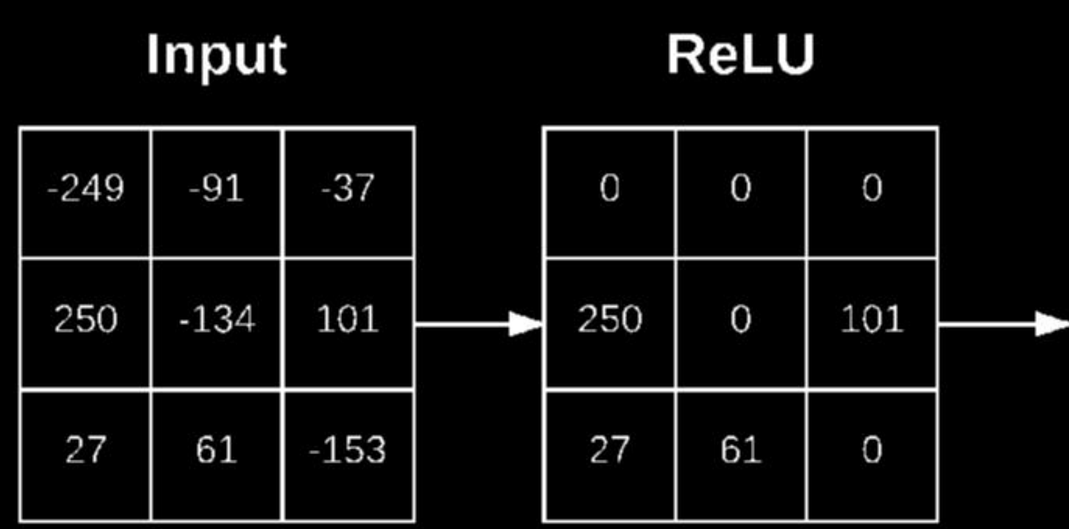

Figure 11.9: An example of an input volume going through a ReLU activation, $max(0, x)$. Activations are done *in-place* so there is no need to create a separate output volume although it is easy to visualize the flow of the network in this manner.

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.

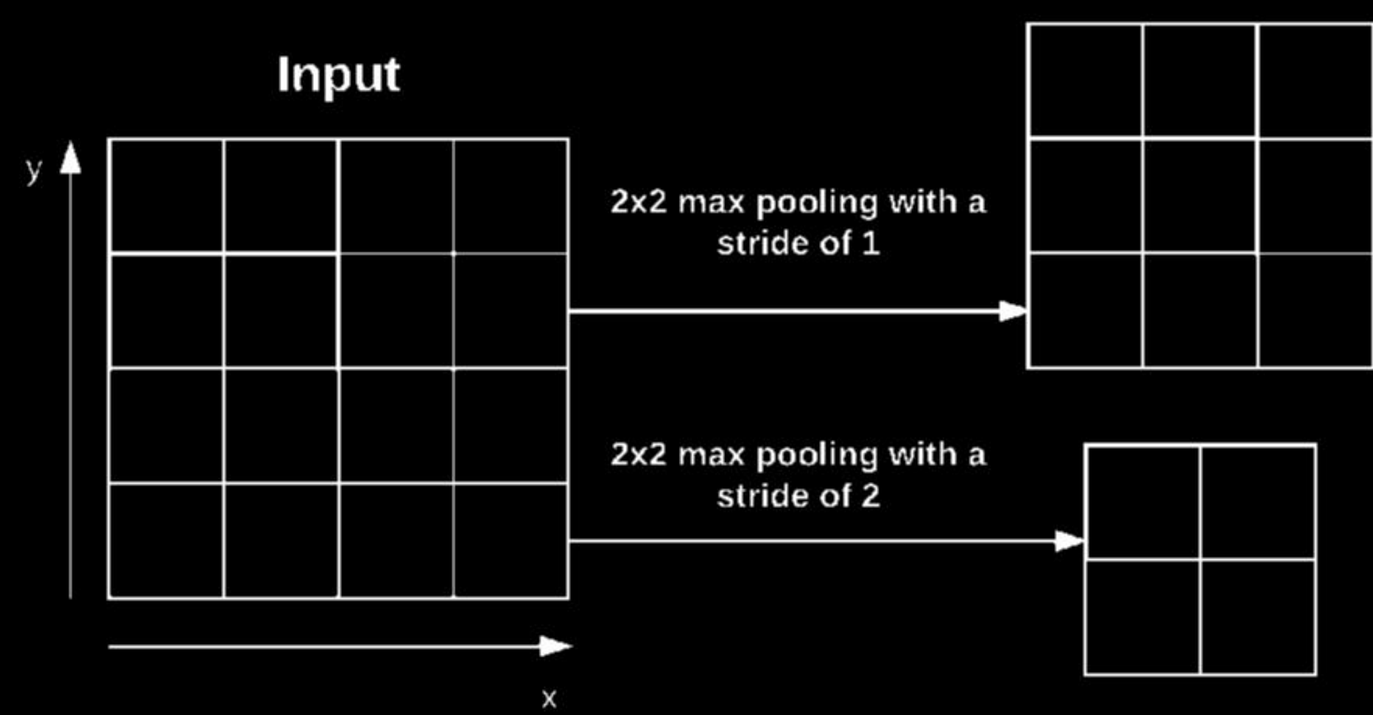

Figure 11.10: **Left:** Our input 4×4 volume. **Right:** Applying 2×2 max pooling with a stride of $S = 1$. **Bottom:** Applying 2×2 max pooling with $S = 2$ – this dramatically reduces the spatial dimensions of our input.

- **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.

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.

You'll learn the basics of:

1. Machine learning
2. Neural Networks
3. Convolutional Neural Networks
4. How to work with your own custom datasets

Volume #2: Practitioner Bundle

The Practitioner Bundle builds on the Starter Bundle and is perfect if you want to study deep learning *in-depth*, understand advanced techniques, and discover common best practices and rules of thumb.

Volume #3: ImageNet Bundle

The ImageNet Bundle is the *complete deep learning for computer vision experience*. In this volume of the book, I demonstrate how to train large-scale neural networks on the *massive* ImageNet dataset as well as tackle real-world case studies, including age + gender prediction, vehicle make + model identification, facial expression recognition, *and much more*.

Need to Upgrade Your Bundle?

If you would ever like to upgrade your bundle, all you have to do is send me a message and we can get the upgrade taken care of ASAP:

http://www.pyimagesearch.com/contact/

Tools of the Trade: Python, Keras, and Mxnet

We'll be utilizing the *Python* programming language for all examples in this book. Python is an extremely easy language to learn. It has intuitive syntax. Is super powerful. And it's **the best way** to work with deep learning algorithms.

The primary deep learning library we'll be using is Keras [1]. The Keras library is maintained by the brilliant François Chollet, a deep learning researcher and engineer at Google. I have been using Keras for *years* and can say that it's hands-down my favorite deep learning package. As a minimal, modular network library that can use *either* Theano or TensorFlow as a backend, you just can't beat Keras.

The second deep learning library we'll be using is mxnet [2] (ImageNet Bundle only), a lightweight, portable, and flexible deep learning library. The mxnet package provides bindings to the Python programming language and specializes in *distributed, multi-machine learning* – the ability to parallelize training across GPUs/devices/nodes is *critical* when training deep neural network architectures on massive datasets (such as ImageNet).

Finally, we'll also be using a few computer vision, image processing, and machine learning libraries such as OpenCV, scikit-image, scikit-learn, etc.

Python, Keras, and mxnet are well-built tools that when combined tighter create a *powerful deep learning development environment* that you can use to master deep learning for visual recognition.

What About TensorFlow?

TensorFlow [3] and Theano [4] are libraries for defining abstract, general-purpose computation graphs. While they are used for deep learning, they are *not* deep learning frameworks and are in fact used for a great many other applications than deep learning.

Keras, on the other hand, *is* a deep learning framework that provides a well-designed API to facilitate building deep neural networks with ease. Under the hood, Keras uses *either* the

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.

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

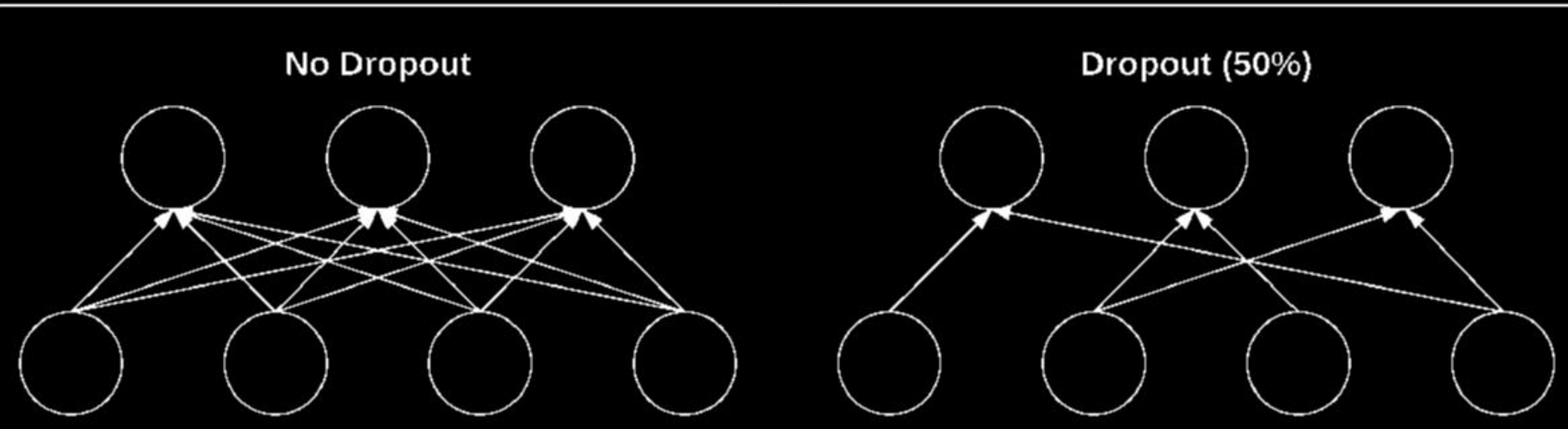

Figure 11.11: **Left:** Two layers of a neural network that are fully-connected with no dropout. **Right:** The same two layers after dropping 50% of the connections.

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).

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.

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.

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*.

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.

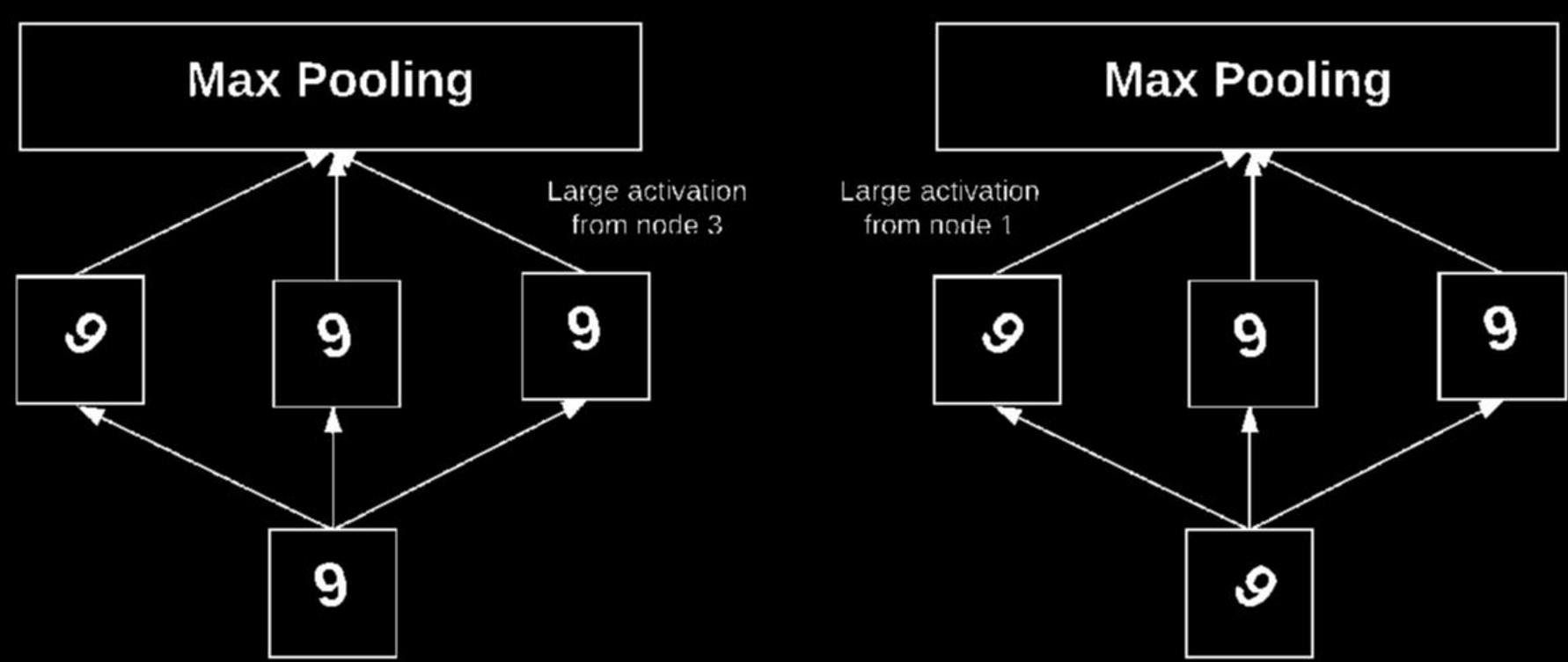

Figure 11.12: CNN as a whole learns filters that will fire when a pattern is presented at a particular orientation. On the *left* the, the digit 9 has been rotated $\approx 10^{\circ}$. This rotation is similar to node three which has learned what the digit 9 looks like when rotated in this manner. This 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.

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.

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.

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":      ,
    "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.

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
```

Deep Learning for Computer Vision with Python

Starter Bundle

Dr. Adrian Rosebrock

1st Edition (1.1.0)

TensorFlow or Theano computational backend, allowing it to take advantage of these powerful computation engines.

Think of a computational backend as an engine that runs in your car. You can replace parts in your engine, optimize others, or replace the engine entirely (provided the engine adheres to a set of specifications of course). Utilizing Keras gives you *portability* across engines and the ability to choose the best engine for your project.

Thinking of this at a different angle, using TensorFlow or Theano to build a deep neural network would be akin to utilizing strictly NumPy to build a machine learning classifier.

Is it possible? Absolutely.

However, it's more beneficial to use a library that is *dedicated* to machine learning, such as scikit-learn [5], rather than reinvent the wheel with NumPy (and at the expense of an order of magnitude more code).

In the same vein, Keras *sits on top* of TensorFlow or Theano, enjoying:

1. The benefits of a powerful underlying computation engine
2. An API that makes it easier for you to build your own deep learning networks

Furthermore, since Keras will be added to the core TensorFlow library at Google [6], we can always integrate TensorFlow code *directly* into our Keras models if we so wish. In many ways, we are getting the best of both worlds by using Keras.

Do I Need to Know OpenCV?

You *do not* need to know the OpenCV computer vision and image processing library [7] to be successful when going through this book.

We only use OpenCV to facilitate basic image processing operations such as loading an image from disk, displaying it to our screen, and a few other basic operations.

That said, a little bit of OpenCV experience goes a long way, so if you're new to OpenCV and computer vision, I *highly recommend* that you work through this book and my other publication, *Practical Python and OpenCV* [8] in tandem.

Remember, deep learning is only *one facet* of computer vision – there are a number of computer vision techniques you should study to round out your knowledge.

Developing Our Own Deep Learning Toolset

One of the inspirations behind writing this book was to demonstrate using *existing deep learning libraries* to build our own custom Python-based toolset, enabling us to train our own deep learning networks.

However, this book isn't just *any* deep learning toolkit ... this toolkit is the *exact same one* I have developed and refined over the past few years doing deep learning research and development of my own.

As we progress through this book, we'll build components of this toolset one at a time. By the end of *Deep Learning for Computer Vision with Python*, our toolset will be able to:

1. Load image datasets from disk, store them in memory, or write them to an optimized database format.
2. Preprocess images such that they are suitable for training a Convolutional Neural Network.
3. Create a blueprint class that can be used to build our own custom implementations of Convolutional Neural Networks.
4. Implement popular CNN architectures by hand, such as AlexNet, VGGNet, GoogLeNet, ResNet, and SqueezeNet (and train them from scratch).
5. *... and much more!*

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

```
1
2       keras.preprocessing.image        img_to_array
3
4        ImageToArrayPreprocessor:
5                  (     , dataFormat=     ):
6
7                .dataFormat = dataFormat
8
9                    (     , image):
10
11
12                   img_to_array(image, data_format=     .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(   ,    )
```

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=    )
```

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

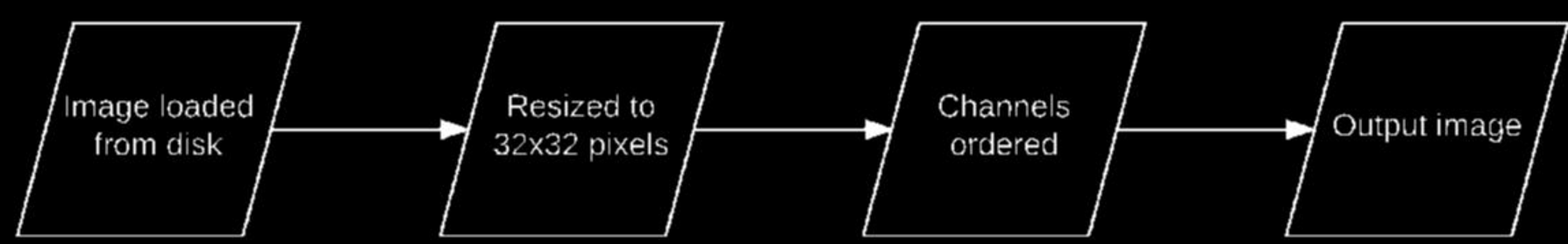

Figure 12.1: An example image pre-processing pipeline that (1) loads an image from disk, (2) resizes it to 32×32 pixels, (3) orders the channel dimensions, and (4) outputs the image.

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*.

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.

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
2       keras.models        Sequential
3       keras.layers.convolutional        Conv2D
```

```
4        keras.layers.core        Activation
5        keras.layers.core        Flatten
6        keras.layers.core        Dense
7        keras        backend    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        ShallowNet:
10
11               (width, height, depth, classes):
12
13
14           model = Sequential()
15           inputShape = (height, width, depth)
16
17
18              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
22           model.add(Conv2D(  , ( ,  ), 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
27          model.add(Flatten())
28          model.add(Dense(classes))
29          model.add(Activation("softmax"))
30
31
32                  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.

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
2       sklearn.preprocessing        LabelBinarizer
3       sklearn.model_selection        train_test_split
4       sklearn.metrics        classification_report
5       pyimagesearch.preprocessing        ImageToArrayPreprocessor
6       pyimagesearch.preprocessing        SimplePreprocessor
7       pyimagesearch.datasets        SimpleDatasetLoader
8       pyimagesearch.nn.conv        ShallowNet
9       keras.optimizers        SGD
10      imutils        paths
11        matplotlib.pyplot    plt
12        numpy    np
13        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
16 ap = argparse.ArgumentParser()
17 ap.add_argument(" d", "  dataset", required=    ,
18      help="path to input dataset")
19 args =     (ap.parse_args())
20
21
22      ("[INFO] loading images...")
23 imagePaths =     (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
26 sp = SimplePreprocessor(  ,   )
27 iap = ImageToArrayPreprocessor()
28
29
30
31 sdl = SimpleDatasetLoader(preprocessors=[sp, iap])
32 (data, labels) = sdl.load(imagePaths, verbose=   )
33 data = data.astype("float") /
```

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
36
37 (trainX, testX, trainY, testY) = train_test_split(data, labels,
38      test_size=    , random_state=  )
39
40
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
45      ("[INFO] compiling model...")
46  opt = SGD(lr=     )
47  model = ShallowNet.build(width=  , height=  , depth= , classes= )
48  model.compile(loss="categorical_crossentropy", optimizer=opt,
49      metrics=["accuracy"])
50
51
52      ("[INFO] training network...")
53  H = model.fit(trainX, trainY, validation_data=(testX, testY),
54      batch_size=  , epochs=   , verbose= )
```

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
57      ("[INFO] evaluating network...")
58  predictions = model.predict(testX, batch_size=  )
59      (classification_report(testY.argmax(axis= ),
60      predictions.argmax(axis= ),
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
64  plt.style.use("ggplot")
65  plt.figure()
66  plt.plot(np.arange( ,    ), H.history["loss"], label="train_loss")
67  plt.plot(np.arange( ,    ), H.history["val_loss"], label="val_loss")
68  plt.plot(np.arange( ,    ), H.history["acc"], label="train_acc")
69  plt.plot(np.arange( ,    ), 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 3000/3000
[INFO] [x] train shape: (2250, 3072) (2250, 3)
[INFO] [x] test shape: (750, 3072) (750, 3)
[INFO] compiling model...
[INFO] epoch=1, [x] time=0.9s
[INFO] [x] 1/1 / 0.9s / 0.9s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=2, [x] time=0.9s
[INFO] [x] 2/2 / 0.9s / 1.8s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=3, [x] time=0.9s
[INFO] [x] 3/3 / 0.9s / 2.7s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=4, [x] time=0.9s
[INFO] [x] 4/4 / 0.9s / 3.6s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=5, [x] time=0.9s
[INFO] [x] 5/5 / 0.9s / 4.5s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=6, [x] time=0.9s
[INFO] [x] 6/6 / 0.9s / 5.4s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=7, [x] time=0.9s
[INFO] [x] 7/7 / 0.9s / 6.3s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=8, [x] time=0.9s
[INFO] [x] 8/8 / 0.9s / 7.2s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=9, [x] time=0.9s
[INFO] [x] 9/9 / 0.9s / 8.1s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=10, [x] time=0.9s
[INFO] [x] 10/10 / 0.9s / 9.0s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=11, [x] time=0.9s
[INFO] [x] 11/11 / 0.9s / 9.9s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=12, [x] time=0.9s
[INFO] [x] 12/12 / 0.9s / 10.8s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=13, [x] time=0.9s
[INFO] [x] 13/13 / 0.9s / 11.7s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=14, [x] time=0.9s
[INFO] [x] 14/14 / 0.9s / 12.6s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=15, [x] time=0.9s
[INFO] [x] 15/15 / 0.9s / 13.5s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=16, [x] time=0.9s
[INFO] [x] 16/16 / 0.9s / 14.4s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=17, [x] time=0.9s
[INFO] [x] 17/17 / 0.9s / 15.3s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=18, [x] time=0.9s
[INFO] [x] 18/18 / 0.9s / 16.2s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=19, [x] time=0.9s
[INFO] [x] 19/19 / 0.9s / 17.1s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=20, [x] time=0.9s
[INFO] [x] 20/20 / 0.9s / 18.0s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=21, [x] time=0.9s
[INFO] [x] 21/21 / 0.9s / 18.9s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=22, [x] time=0.9s
[INFO] [x] 22/22 / 0.9s / 19.8s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=23, [x] time=0.9s
[INFO] [x] 23/23 / 0.9s / 20.7s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=24, [x] time=0.9s
[INFO] [x] 24/24 / 0.9s / 21.6s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=25, [x] time=0.9s
[INFO] [x] 25/25 / 0.9s / 22.5s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=26, [x] time=0.9s
[INFO] [x] 26/26 / 0.9s / 23.4s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=27, [x] time=0.9s
[INFO] [x] 27/27 / 0.9s / 24.3s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=28, [x] time=0.9s
[INFO] [x] 28/28 / 0.9s / 25.2s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=29, [x] time=0.9s
[INFO] [x] 29/29 / 0.9s / 26.1s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=30, [x] time=0.9s
[INFO] [x] 30/30 / 0.9s / 27.0s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=31, [x] time=0.9s
[INFO] [x] 31/31 / 0.9s / 27.9s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=32, [x] time=0.9s
[INFO] [x] 32/32 / 0.9s / 28.8s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=33, [x] time=0.9s
[INFO] [x] 33/33 / 0.9s / 29.7s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=34, [x] time=0.9s
[INFO] [x] 34/34 / 0.9s / 30.6s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=35, [x] time=0.9s
[INFO] [x] 35/35 / 0.9s / 31.5s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=36, [x] time=0.9s
[INFO] [x] 36/36 / 0.9s / 32.4s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=37, [x] time=0.9s
[INFO] [x] 37/37 / 0.9s / 33.3s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=38, [x] time=0.9s
[INFO] [x] 38/38 / 0.9s / 34.2s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=39, [x] time=0.9s
[INFO] [x] 39/39 / 0.9s / 35.1s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=40, [x] time=0.9s
[INFO] [x] 40/40 / 0.9s / 36.0s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=41, [x] time=0.9s
[INFO] [x] 41/41 / 0.9s / 36.9s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=42, [x] time=0.9s
[INFO] [x] 42/42 / 0.9s / 37.8s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=43, [x] time=0.9s
[INFO] [x] 43/43 / 0.9s / 38.7s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=44, [x] time=0.9s
[INFO] [x] 44/44 / 0.9s / 39.6s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=45, [x] time=0.9s
[INFO] [x] 45/45 / 0.9s / 40.5s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=46, [x] time=0.9s
[INFO] [x] 46/46 / 0.9s / 41.4s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=47, [x] time=0.9s
[INFO] [x] 47/47 / 0.9s / 42.3s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=48, [x] time=0.9s
[INFO] [x] 48/48 / 0.9s / 43.2s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=49, [x] time=0.9s
[INFO] [x] 49/49 / 0.9s / 44.1s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=50, [x] time=0.9s
[INFO] [x] 50/50 / 0.9s / 45.0s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=51, [x] time=0.9s
[INFO] [x] 51/51 / 0.9s / 45.9s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=52, [x] time=0.9s
[INFO] [x] 52/52 / 0.9s / 46.8s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=53, [x] time=0.9s
[INFO] [x] 53/53 / 0.9s / 47.7s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=54, [x] time=0.9s
[INFO] [x] 54/54 / 0.9s / 48.6s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=55, [x] time=0.9s
[INFO] [x] 55/55 / 0.9s / 49.5s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=56, [x] time=0.9s
[INFO] [x] 56/56 / 0.9s / 50.4s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=57, [x] time=0.9s
[INFO] [x] 57/57 / 0.9s / 51.3s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=58, [x] time=0.9s
[INFO] [x] 58/58 / 0.9s / 52.2s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=59, [x] time=0.9s
[INFO] [x] 59/59 / 0.9s / 53.1s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=60, [x] time=0.9s
[INFO] [x] 60/60 / 0.9s / 54.0s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=61, [x] time=0.9s
[INFO] [x] 61/61 / 0.9s / 54.9s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=62, [x] time=0.9s
[INFO] [x] 62/62 / 0.9s / 55.8s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=63, [x] time=0.9s
[INFO] [x] 63/63 / 0.9s / 56.7s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=64, [x] time=0.9s
[INFO] [x] 64/64 / 0.9s / 57.6s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=65, [x] time=0.9s
[INFO] [x] 65/65 / 0.9s / 58.5s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=66, [x] time=0.9s
[INFO] [x] 66/66 / 0.9s / 59.4s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=67, [x] time=0.9s
[INFO] [x] 67/67 / 0.9s / 60.3s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=68, [x] time=0.9s
[INFO] [x] 68/68 / 0.9s / 61.2s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=69, [x] time=0.9s
[INFO] [x] 69/69 / 0.9s / 62.1s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=70, [x] time=0.9s
[INFO] [x] 70/70 / 0.9s / 63.0s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=71, [x] time=0.9s
[INFO] [x] 71/71 / 0.9s / 63.9s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=72, [x] time=0.9s
[INFO] [x] 72/72 / 0.9s / 64.8s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=73, [x] time=0.9s
[INFO] [x] 73/73 / 0.9s / 65.7s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=74, [x] time=0.9s
[INFO] [x] 74/74 / 0.9s / 66.6s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=75, [x] time=0.9s
[INFO] [x] 75/75 / 0.9s / 67.5s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=76, [x] time=0.9s
[INFO] [x] 76/76 / 0.9s / 68.4s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=77, [x] time=0.9s
[INFO] [x] 77/77 / 0.9s / 69.3s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=78, [x] time=0.9s
[INFO] [x] 78/78 / 0.9s / 70.2s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=79, [x] time=0.9s
[INFO] [x] 79/79 / 0.9s / 71.1s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=80, [x] time=0.9s
[INFO] [x] 80/80 / 0.9s / 72.0s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=81, [x] time=0.9s
[INFO] [x] 81/81 / 0.9s / 72.9s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=82, [x] time=0.9s
[INFO] [x] 82/82 / 0.9s / 73.8s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=83, [x] time=0.9s
[INFO] [x] 83/83 / 0.9s / 74.7s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=84, [x] time=0.9s
[INFO] [x] 84/84 / 0.9s / 75.6s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=85, [x] time=0.9s
[INFO] [x] 85/85 / 0.9s / 76.5s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=86, [x] time=0.9s
[INFO] [x] 86/86 / 0.9s / 77.4s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=87, [x] time=0.9s
[INFO] [x] 87/87 / 0.9s / 78.3s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=88, [x] time=0.9s
[INFO] [x] 88/88 / 0.9s / 79.2s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=89, [x] time=0.9s
[INFO] [x] 89/89 / 0.9s / 80.1s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=90, [x] time=0.9s
[INFO] [x] 90/90 / 0.9s / 81.0s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=91, [x] time=0.9s
[INFO] [x] 91/91 / 0.9s / 81.9s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=92, [x] time=0.9s
[INFO] [x] 92/92 / 0.9s / 82.8s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=93, [x] time=0.9s
[INFO] [x] 93/93 / 0.9s / 83.7s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=94, [x] time=0.9s
[INFO] [x] 94/94 / 0.9s / 84.6s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=95, [x] time=0.9s
[INFO] [x] 95/95 / 0.9s / 85.5s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=96, [x] time=0.9s
[INFO] [x] 96/96 / 0.9s / 86.4s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=97, [x] time=0.9s
[INFO] [x] 97/97 / 0.9s / 87.3s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=98, [x] time=0.9s
[INFO] [x] 98/98 / 0.9s / 88.2s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=99, [x] time=0.9s
[INFO] [x] 99/99 / 0.9s / 89.1s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] epoch=100, [x] time=0.9s
[INFO] [x] 100/100 / 0.9s / 90.0s
[INFO] [x] train loss=1.09861229 acc=0.3333
[INFO] [x] test loss=1.09861229 acc=0.3333
[INFO] [x] time=90.0s
```

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.

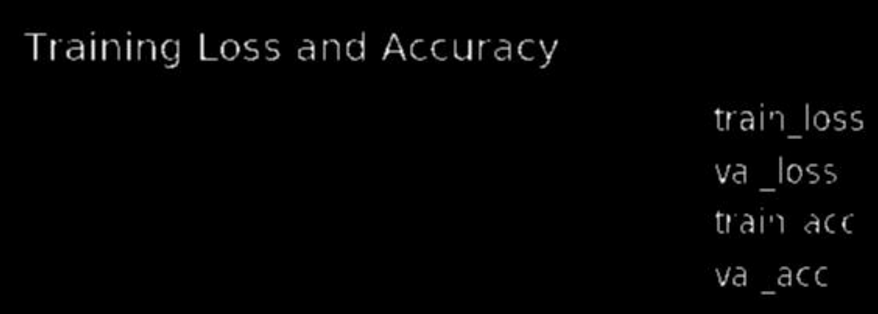

Figure 12.2: A plot of our loss and accuracy over the course of 100 epochs for the ShallowNet architecture trained on the Animals dataset.

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%!

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:

```

sklearn.preprocessing        LabelBinarizer
sklearn.metrics        classification_report
pyimagesearch.nn.conv        ShallowNet
keras.optimizers        SGD
keras.datasets        cifar10
  matplotlib.pyplot    plt
    numpy    np



    ("[INFO] loading CIFAR 10 data...")
((trainX, trainY), (testX, testY)) = cifar10.load_data()
trainX = trainX.astype("float") /
testX = testX.astype("float") /


lb = LabelBinarizer()
```

```
19 trainY = lb.fit_transform(trainY)
20 testY = lb.transform(testY)
21
22
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
27      ("[INFO] compiling model...")
28 opt = SGD(lr=    )
29 model = ShallowNet.build(width=  , height=  , depth= , classes=  )
30 model.compile(loss="categorical_crossentropy", optimizer=opt,
31      metrics=["accuracy"])
32
33
34      ("[INFO] training network...")
35 H = model.fit(trainX, trainY, validation_data=(testX, testY),
36      batch_size=  , epochs=  , verbose= )
```

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
39      ("[INFO] evaluating network...")
40 predictions = model.predict(testX, batch_size=  )
41      (classification_report(testY.argmax(axis= ),
42      predictions.argmax(axis= ), 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
45 plt.style.use("ggplot")
46 plt.figure()
47 plt.plot(np.arange( ,   ), H.history["loss"], label="train_loss")
48 plt.plot(np.arange( ,   ), H.history["val_loss"], label="val_loss")
```

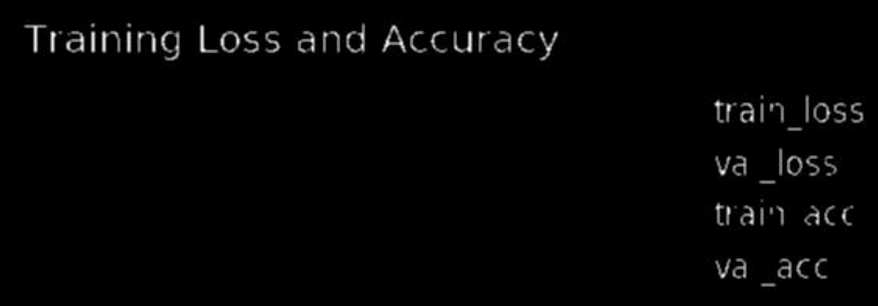

Figure 12.3: Loss and accuracy for ShallowNet trained on CIFAR-10. Our network obtains 60% classification accuracy; however, it is overfitting. Further accuracy can be obtained by applying regularization, which we'll cover later in this book.

```
plt.plot(np.arange( ,   ), H.history["acc"], label="train_acc")
plt.plot(np.arange( ,   ), 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 ShallowNet on CIFAR-10, simply execute the following command:

```
python shallownet_cifar10.py
```

Summary

We are living in a special time in machine learning, neural network, and deep learning history. Never in the history of machine learning and neural networks have the available tools at our disposable been so exceptional.

From a software perspective, we have libraries such as Keras and mxnet complete with Python bindings, enabling us to rapidly construct deep learning architectures in a *fraction of the time* it took us just years before.

Then, from a hardware view, general purpose GPUs are becoming *increasingly cheaper* while *still becoming more powerful*. The advances in GPU technology alone have allowed anyone with a modest budget to build even a simple gaming rig to conduct *meaningful* deep learning research.

Deep learning is an *exciting field*, and due to these powerful GPUs, modular libraries, and unbridled, intelligent researchers, we're seeing new publications that push the state-of-the-art coming on a *monthly basis*.

You see, *now* is the time to undertake studying deep learning for computer vision.

Don't miss out on this time in history – not only should you be a part of deep learning, but those who capitalize early are sure to see immense returns on their investment of time, resources, and creativity.

Enjoy this book. I'm excited to see where it takes you in this amazing field.

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.

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.

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:

```

sklearn.preprocessing        LabelBinarizer
sklearn.model_selection        train_test_split
sklearn.metrics        classification_report
pyimagesearch.preprocessing        ImageToArrayPreprocessor
pyimagesearch.preprocessing        SimplePreprocessor
pyimagesearch.datasets        SimpleDatasetLoader
```

```
8       pyimagesearch.nn.conv        ShallowNet
9       keras.optimizers        SGD
10      imutils        paths
11        matplotlib.pyplot    plt
12        numpy    np
13        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
16  ap = argparse.ArgumentParser()
17  ap.add_argument(" d", "  dataset", required=    ,
18       help="path to input dataset")
19  ap.add_argument(" m", "  model", required=    ,
20       help="path to output model")
21  args =     (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
24       ("[INFO] loading images...")
25  imagePaths =     (paths.list_images(args["dataset"]))
26
27
28  sp = SimplePreprocessor(  ,   )
29  iap = ImageToArrayPreprocessor()
30
31
32
33  sdl = SimpleDatasetLoader(preprocessors=[sp, iap])
34  (data, labels) = sdl.load(imagePaths, verbose=   )
35  data = data.astype("float") /
```

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

```
37
38
39  (trainX, testX, trainY, testY) = train_test_split(data, labels,
40       test_size=    , random_state=  )
41
42
```

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

Training ShallowNet is handled via the code block below:

```

     ("[INFO] compiling model...")
opt = SGD(lr=     )
model = ShallowNet.build(width=  , height=  , depth= , classes= )
model.compile(loss="categorical_crossentropy", optimizer=opt,
     metrics=["accuracy"])


     ("[INFO] training network...")
H = model.fit(trainX, trainY, validation_data=(testX, testY),
     batch_size=  , epochs=   , verbose= )
```

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:

```

     ("[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:

```

     ("[INFO] evaluating network...")
predictions = model.predict(testX, batch_size=  )
     (classification_report(testY.argmax(axis= ),
     predictions.argmax(axis= ),
     target_names=["cat", "dog", "panda"]))
```

As well as plot our loss and accuracy:

```

plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange( ,    ), H.history["loss"], label="train_loss")
plt.plot(np.arange( ,    ), H.history["val_loss"], label="val_loss")
plt.plot(np.arange( ,    ), H.history["acc"], label="train_acc")
plt.plot(np.arange( ,    ), 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
```

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.

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
2       pyimagesearch.preprocessing        ImageToArrayPreprocessor
3       pyimagesearch.preprocessing        SimplePreprocessor
4       pyimagesearch.datasets        SimpleDatasetLoader
5       keras.models        load_model
6       imutils        paths
7         numpy    np
8         argparse
9         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
12 ap = argparse.ArgumentParser()
13 ap.add_argument(" d", "  dataset", required=    ,
14      help="path to input dataset")
15 ap.add_argument(" m", "  model", required=    ,
16      help="path to pre trained model")
17 args =     (ap.parse_args())
18
19
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
23
24      ("[INFO] sampling images...")
25  imagePaths = np.array(    (paths.list_images(args["dataset"])))
26  idxs = np.random.randint( ,    (imagePaths), size=(  ,))
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
30  sp = SimplePreprocessor(  ,   )
31  iap = ImageToArrayPreprocessor()
32
33
34
35  sdl = SimpleDatasetLoader(preprocessors=[sp, iap])
36  (data, labels) = sdl.load(imagePaths)
37  data = data.astype("float") /
```

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
40      ("[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
44      ("[INFO] predicting...")
45  preds = model.predict(data, batch_size=  ).argmax(axis= )
```

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
48      (i, imagePath)              (imagePaths):
49
50
51       image = cv2.imread(imagePath)
52       cv2.putText(image, "Label: {}".format(classLabels[preds[i]]),
53            (  ,   ), cv2.FONT_HERSHEY_SIMPLEX,    , ( ,    ,  ),  )
54       cv2.imshow("Image", image)
55       cv2.waitKey( )
```

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
```

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.

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*.

2. What Is Deep Learning?

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

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

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

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

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

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

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.

The LeNet Architecture

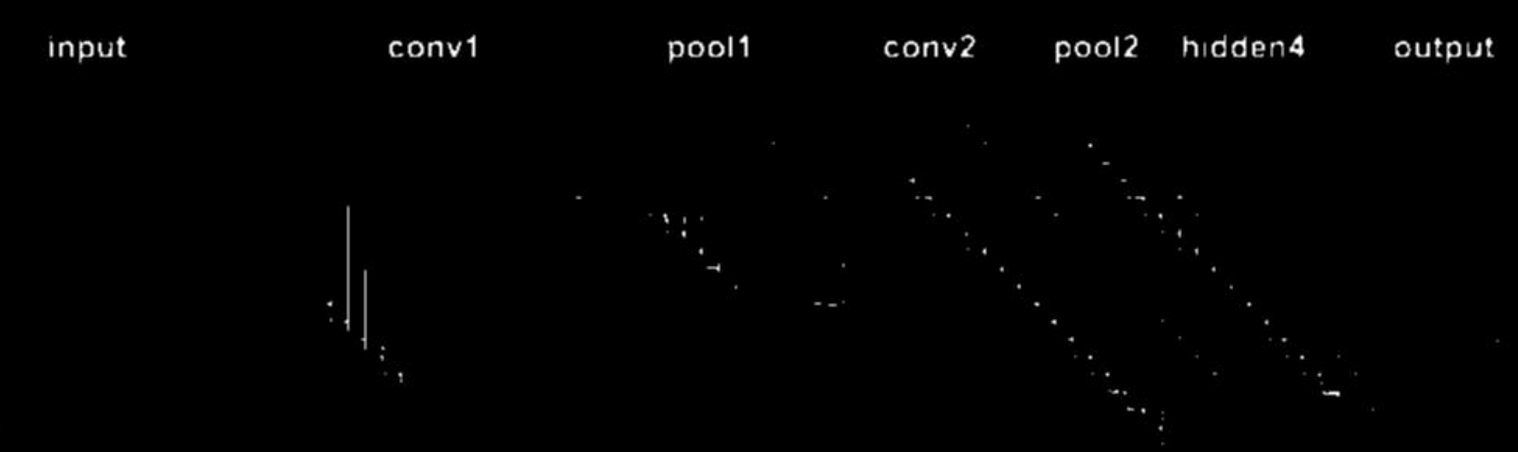

Figure 14.1: The LeNet architecture consists of two series of `CONV => TANH => POOL` layer sets followed by a fully-connected layer and softmax output. *Photo Credit:*

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.

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
2       keras.models        Sequential
3       keras.layers.convolutional        Conv2D
4       keras.layers.convolutional        MaxPooling2D
5       keras.layers.core        Activation
6       keras.layers.core        Flatten
7       keras.layers.core        Dense
8       keras        backend    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      LeNet:
11
12              (width, height, depth, classes):
13
14          model = Sequential()
15          inputShape = (height, width, depth)
16
17
18             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:

```

        model.add(Conv2D(  , ( ,  ), padding="same",
             input_shape=inputShape))
        model.add(Activation("relu"))
        model.add(MaxPooling2D(pool_size=( ,  ), strides=( ,  )))
```

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:

```

        model.add(Conv2D(  , ( ,  ), padding="same"))
        model.add(Activation("relu"))
        model.add(MaxPooling2D(pool_size=( ,  ), strides=( ,  )))
```

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

```

        model.add(Flatten())
        model.add(Dense(   ))
        model.add(Activation("relu"))
```

Followed by the final softmax classifier:

```

        model.add(Dense(classes))
        model.add(Activation("softmax"))


                model
```

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

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:

```

    pyimagesearch.nn.conv        LeNet
```

```
3       keras.optimizers        SGD
4       sklearn.preprocessing        LabelBinarizer
5       sklearn.model_selection        train_test_split
6       sklearn.metrics        classification_report
7       sklearn        datasets
8       keras        backend    K
9         matplotlib.pyplot    plt
10        numpy    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
13
14        ("[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
19
20
21      K.image_data_format() == "channels_first":
22        data = data.reshape(data.shape[ ],   ,   ,   )
23
24
25
26      :
27        data = data.reshape(data.shape[ ],   ,   ,  )
```

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
30
31 (trainX, testX, trainY, testY) = train_test_split(data /      ,
32      dataset.target.astype("int"), test_size=    , random_state=  )
33
34
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
40      ("[INFO] compiling model...")
41 opt = SGD(lr=    )
42 model = LeNet.build(width=  , height=  , depth= , classes=  )
43 model.compile(loss="categorical_crossentropy", optimizer=opt,
44      metrics=["accuracy"])
45
46
47      ("[INFO] training network...")
48 H = model.fit(trainX, trainY, validation_data=(testX, testY),
49      batch_size=   , epochs=  , verbose= )
```

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
52      ("[INFO] evaluating network...")
53 predictions = model.predict(testX, batch_size=   )
54      (classification_report(testY.argmax(axis= ),
55      predictions.argmax(axis= ),
56      target_names=[   (x)     x    le.classes_]))
57
```

```
58
59 plt.style.use("ggplot")
60 plt.figure()
61 plt.plot(np.arange( ,   ), H.history["loss"], label="train_loss")
62 plt.plot(np.arange( ,   ), H.history["val_loss"], label="val_loss")
63 plt.plot(np.arange( ,   ), H.history["acc"], label="train_acc")
64 plt.plot(np.arange( ,   ), 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:

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.

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.

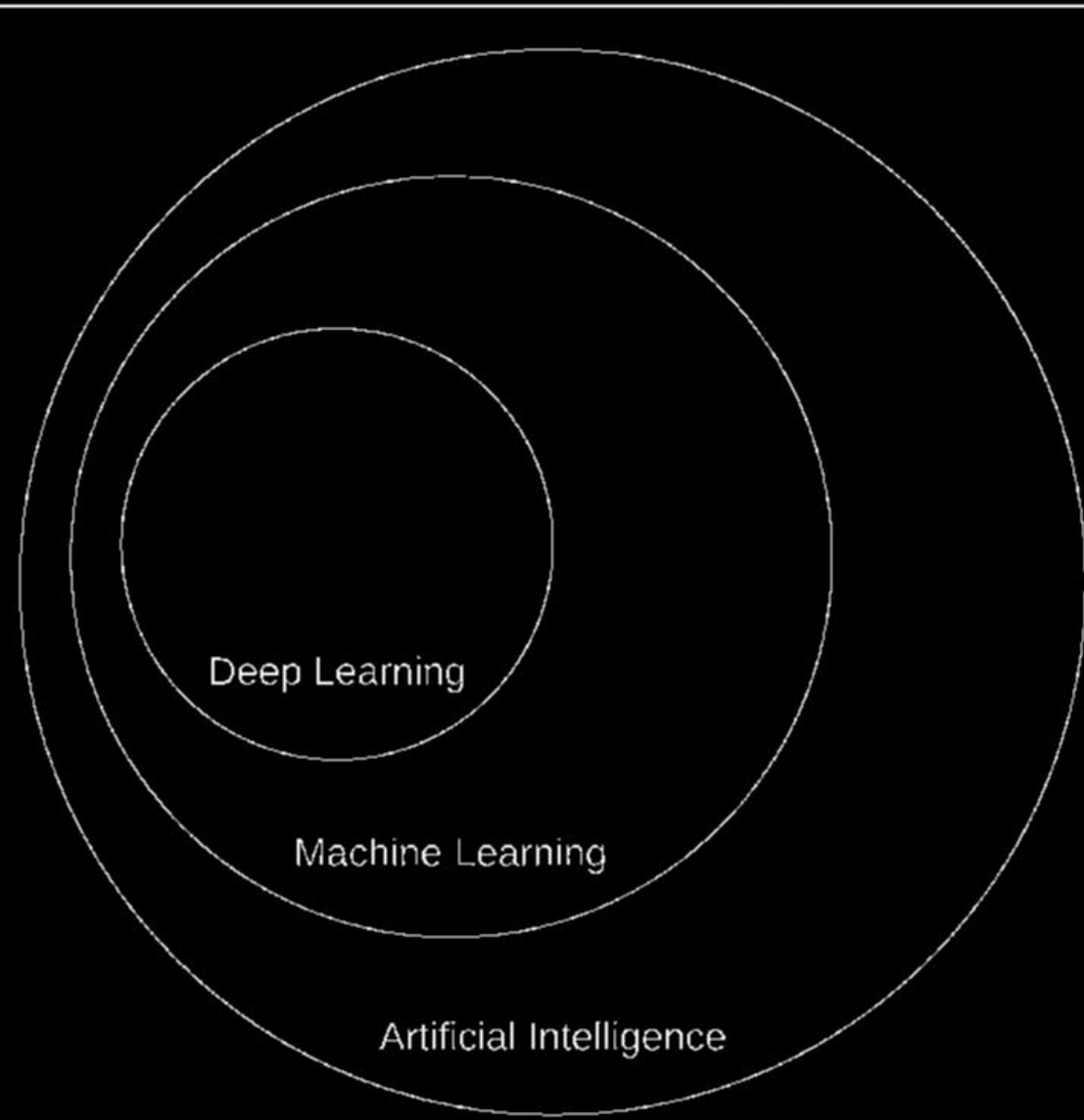

Figure 2.1: A Venn diagram describing deep learning as a subfield of machine learning which is in turn a subfield of artificial intelligence (Image inspired by Figure 1.4 of Goodfellow et al. [10]).

A Concise History of Neural Networks and Deep Learning

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

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

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

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

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

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*.

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*.

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.

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:

```
from keras.models import Sequential
from keras.layers.normalization import BatchNormalization
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       keras.layers.convolutional        MaxPooling2D
6       keras.layers.core        Activation
7       keras.layers.core        Flatten
8       keras.layers.core        Dropout
9       keras.layers.core        Dense
10      keras        backend    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      MiniVGGNet:
13
14              (width, height, depth, classes):
15
16
17          model = Sequential()
18          inputShape = (height, width, depth)
19          chanDim = -
20
21
22
23              K.image_data_format() == "channels_first":
24                inputShape = (depth, height, width)
25                chanDim =
```

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
28          model.add(Conv2D(  , ( ,  ), padding="same",
29               input_shape=inputShape))
30          model.add(Activation("relu"))
31          model.add(BatchNormalization(axis=chanDim))
32          model.add(Conv2D(  , ( ,  ), padding="same"))
33          model.add(Activation("relu"))
34          model.add(BatchNormalization(axis=chanDim))
35          model.add(MaxPooling2D(pool_size=( ,  )))
36          model.add(Dropout(    ))
```

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
39          model.add(Conv2D(  , ( ,  ), padding="same"))
40          model.add(Activation("relu"))
41          model.add(BatchNormalization(axis=chanDim))
42          model.add(Conv2D(  , ( ,  ), padding="same"))
43          model.add(Activation("relu"))
44          model.add(BatchNormalization(axis=chanDim))
45          model.add(MaxPooling2D(pool_size=( ,  )))
46          model.add(Dropout(    ))
```

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
49          model.add(Flatten())
50          model.add(Dense(   ))
51          model.add(Activation("relu"))
52          model.add(BatchNormalization())
53          model.add(Dropout(   ))
```

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
56          model.add(Dense(classes))
57          model.add(Activation("softmax"))
58
59
60                  model
```

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

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
2         matplotlib
3  matplotlib.use("Agg")
4
5
6       sklearn.preprocessing        LabelBinarizer
7       sklearn.metrics        classification_report
8       pyimagesearch.nn.conv        MiniVGGNet
9       keras.optimizers        SGD
10      keras.datasets        cifar10
11        matplotlib.pyplot    plt
12        numpy    np
13        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
16  ap = argparse.ArgumentParser()
17  ap.add_argument(" o", "  output", required=    ,
18       help="path to the output loss/accuracy plot")
19  args =     (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
22
23       ("[INFO] loading CIFAR 10 data...")
24  ((trainX, trainY), (testX, testY)) = cifar10.load_data()
```

```
trainX = trainX.astype("float") /
testX = testX.astype("float") /


lb = LabelBinarizer()
trainY = lb.fit_transform(trainY)
testY = lb.transform(testY)


labelNames = ["airplane", "automobile", "bird", "cat", "deer",
     "dog", "frog", "horse", "ship", "truck"]
```

Let's compile our model and start training MiniVGGNet:

```

     ("[INFO] compiling model...")
opt = SGD(lr=    , decay=    /   , momentum=   , nesterov=    )
model = MiniVGGNet.build(width=  , height=  , depth= , classes=  )
model.compile(loss="categorical_crossentropy", optimizer=opt,
     metrics=["accuracy"])


     ("[INFO] training network...")
H = model.fit(trainX, trainY, validation_data=(testX, testY),
     batch_size=  , epochs=  , verbose= )
```

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:

```

     ("[INFO] evaluating network...")
predictions = model.predict(testX, batch_size=  )
     (classification_report(testY.argmax(axis= ),
     predictions.argmax(axis= ), target_names=labelNames))
```

And with save our loss and accuracy plot to disk:

```

plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange( ,   ), H.history["loss"], label="train_loss")
```

```
plt.plot(np.arange( ,   ), H.history["val_loss"], label="val_loss")
plt.plot(np.arange( ,   ), H.history["acc"], label="train_acc")
plt.plot(np.arange( ,   ), H.history["val_acc"], label="val_acc")
plt.title("Training Loss and Accuracy on CIFAR 10")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
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.

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
```

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.

Without Batch Normalization

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

```
27
28          model.add(Conv2D(  , ( ,  ), padding="same",
29               input_shape=inputShape))
30          model.add(Activation("relu"))
31
32          model.add(Conv2D(  , ( ,  ), padding="same"))
33          model.add(Activation("relu"))
34
35          model.add(MaxPooling2D(pool_size=( ,  )))
36          model.add(Dropout(    ))
```

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
```

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:

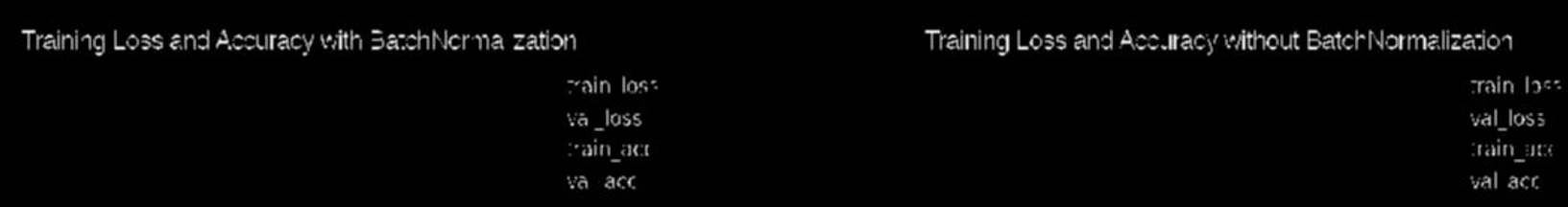

Figure 15.1: **Left:** MiniVGGNet trained on CIFAR-10 *with* batch normalization. **Right:** MiniVGGNet trained on CIFAR-10 *without* batch normalization. Applying batch normalization allows us to obtain higher classification accuracy and reduce the affects of overfitting.

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.

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

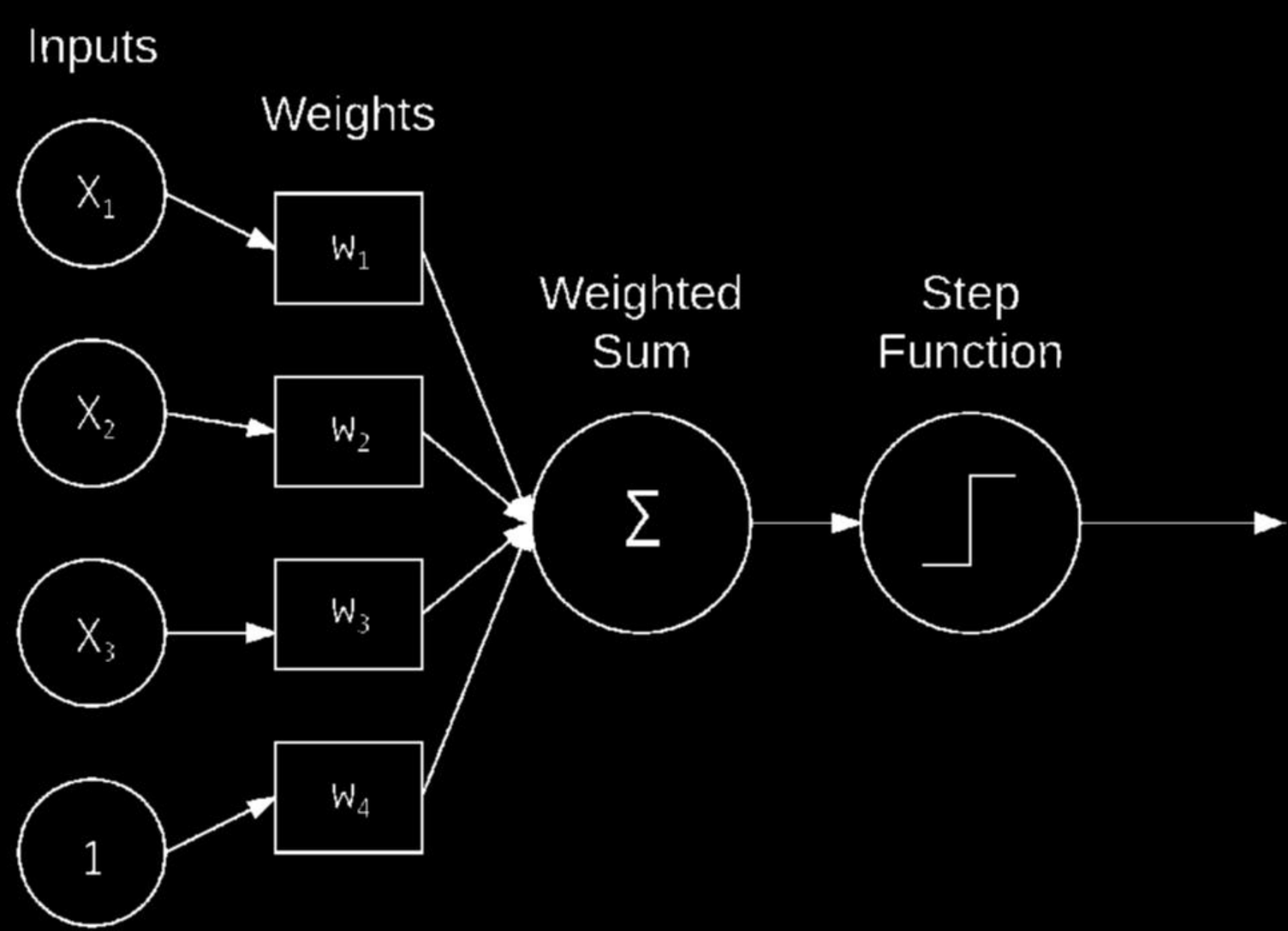

Figure 2.2: An example of the simple Perceptron network architecture that accepts a number of inputs, computes a weighted sum, and applies a step function to obtain the final prediction. We'll review the Perceptron in detail inside Chapter 10.

stars from the red circles.

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

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

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

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

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

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

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.

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.

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:

```

    ("[INFO] compiling model...")
opt = SGD(lr=    , decay=    /   , momentum=   , nesterov=   )
model = MiniVGGNet.build(width=  , height=  , depth= , classes=  )
model.compile(loss="categorical_crossentropy", optimizer=opt,
    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.

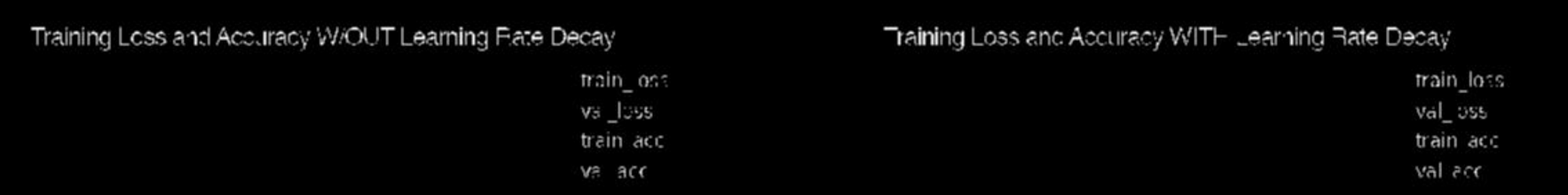

Figure 16.1: **Left:** Training MiniVGGNet on CIFAR-10 *without* learning rate decay. Notice how loss starts to increase past epoch 25, indicating that overfitting is happening. **Right:** Applying a decay factor of $0.01/40$. This reduces the learning rate over time, helping alleivate the affects of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

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.

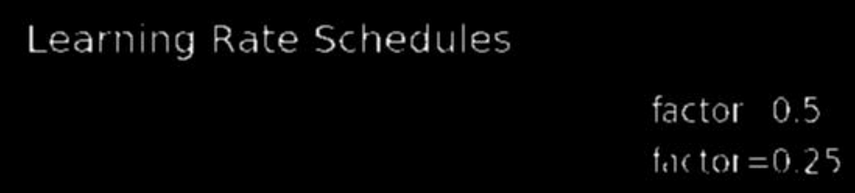

Figure 16.2: An example of two learning rate schedules that drop the learning rate in a piecewise fashion. Lowering the *factor* value increases the speed of the drop. In each case, the learning rate approaches zero at the final epoch.

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.

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((  + 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
2         matplotlib
3   matplotlib.use("Agg")
4
5
6       sklearn.preprocessing        LabelBinarizer
7       sklearn.metrics        classification_report
8       pyimagesearch.nn.conv        MiniVGGNet
9       keras.callbacks        LearningRateScheduler
10      keras.optimizers        SGD
11      keras.datasets        cifar10
12        matplotlib.pyplot    plt
13        numpy    np
14        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                (epoch):
17
18
19      initAlpha =
20      factor =
21      dropEvery =
22
23
24      alpha = initAlpha * (factor ** np.floor((  + epoch) / dropEvery))
25
26
27                (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
30  ap = argparse.ArgumentParser()
31  ap.add_argument(" o", "  output", required=    ,
32       help="path to the output loss/accuracy plot")
33  args =     (ap.parse_args())
34
35
36
37       ("[INFO] loading CIFAR 10 data...")
38  ((trainX, trainY), (testX, testY)) = cifar10.load_data()
39  trainX = trainX.astype("float") /
40  testX = testX.astype("float") /
41
42
43  lb = LabelBinarizer()
44  trainY = lb.fit_transform(trainY)
45  testY = lb.transform(testY)
46
47
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
52
53  callbacks = [LearningRateScheduler(step_decay)]
54
55
56  opt = SGD(lr=    , momentum=   , nesterov=    )
57  model = MiniVGGNet.build(width=  , height=  , depth= , classes=  )
58  model.compile(loss="categorical_crossentropy", optimizer=opt,
59       metrics=["accuracy"])
60
61
62  H = model.fit(trainX, trainY, validation_data=(testX, testY),
63       batch_size=  , epochs=  , callbacks=callbacks, verbose= )
```

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
66        ("[INFO] evaluating network...")
67  predictions = model.predict(testX, batch_size=  )
68        (classification_report(testY.argmax(axis= ),
69        predictions.argmax(axis= ), target_names=labelNames))
```

As well as plot the loss and accuracy:

```
71
72  plt.style.use("ggplot")
73  plt.figure()
74  plt.plot(np.arange( ,   ), H.history["loss"], label="train_loss")
75  plt.plot(np.arange( ,   ), H.history["val_loss"], label="val_loss")
76  plt.plot(np.arange( ,   ), H.history["acc"], label="train_acc")
77  plt.plot(np.arange( ,   ), 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
```

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.

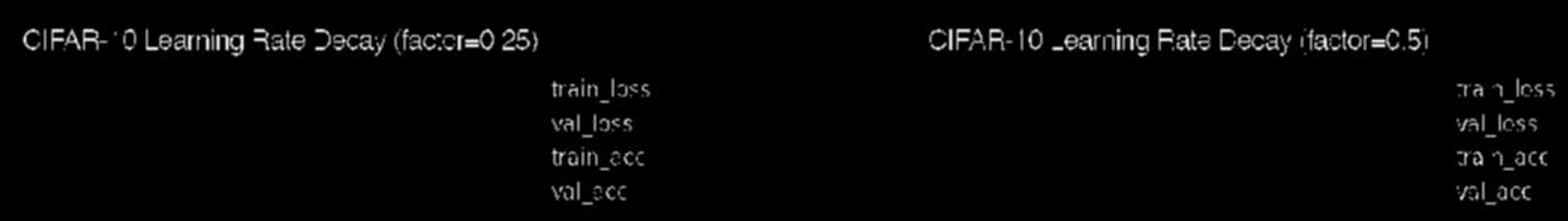

Figure 16.3: **Left:** Plotting the accuracy/loss of our network using a faster learning rate drop with a factor of 0.25. Notice how loss/accuracy stagnate past epoch 15 as the learning rate is too small. **Right:** Accuracy/loss of our network with a slower learning rate drop ($factor = 0.5$). This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

```
                (epoch):

    initAlpha =
    factor =
    dropEvery =
```

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
```

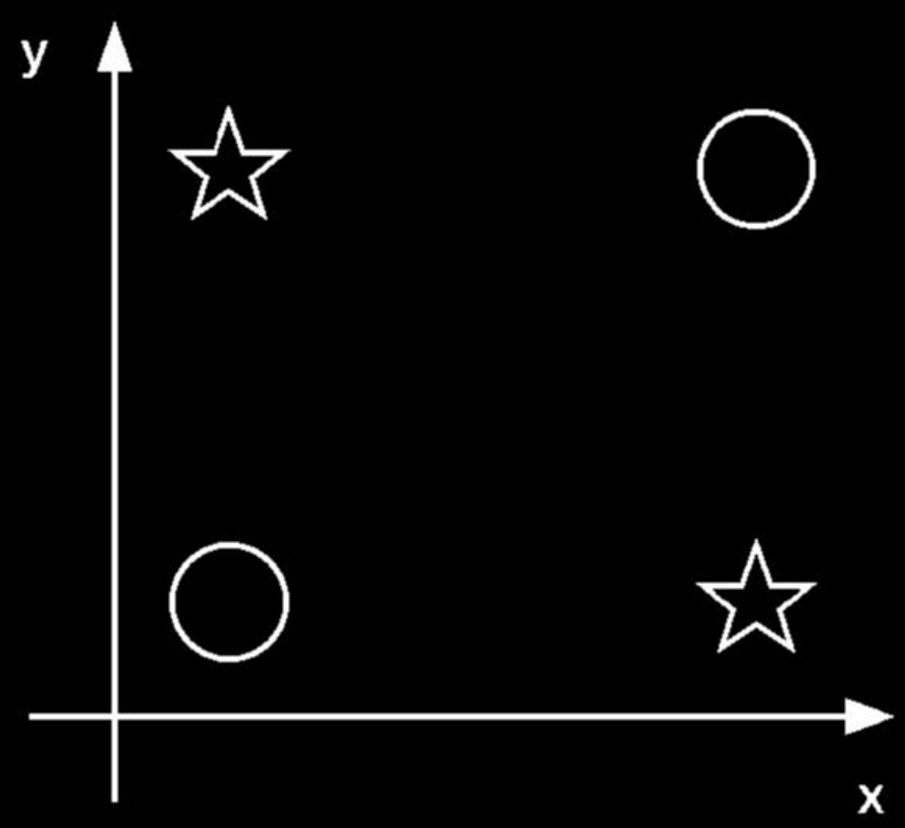

Figure 2.3: The XOR (E(X)clusive Or) dataset is an example of a nonlinear separable problem that the Perceptron *cannot* solve. Take a second to convince yourself that it is impossible to draw a single line that separates the blue stars from the red circles.

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

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

Hierarchical Feature Learning

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

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

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

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.

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.

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:

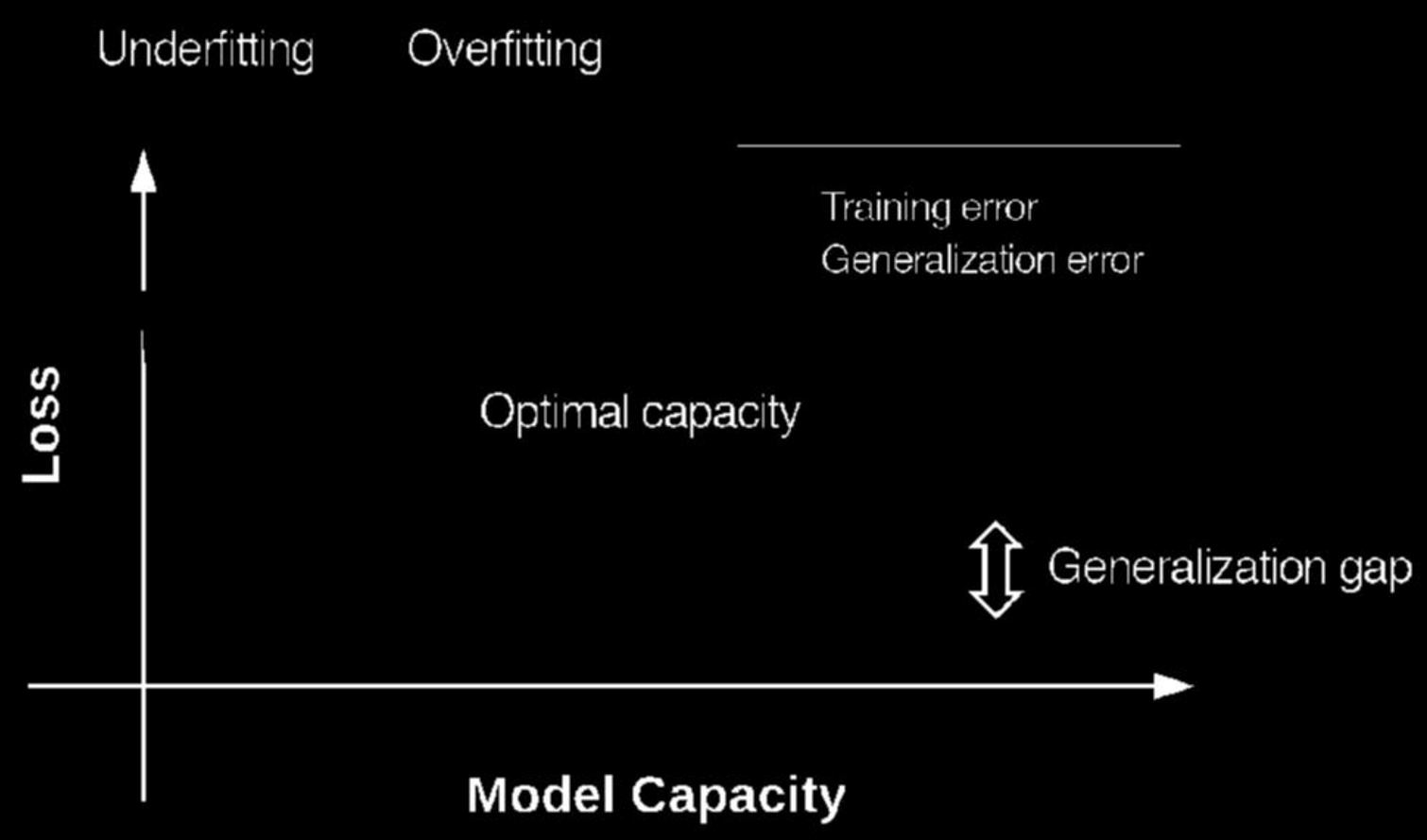

Figure 17.1: Relationship between model capacity and loss. The vertical purple line separates optimal capacity from underfitting (left) and overfitting (right). When we are underfitting the generalization gap is maintained. Optimal capacity occurs when the loss for both training and generalization levels 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.

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]).

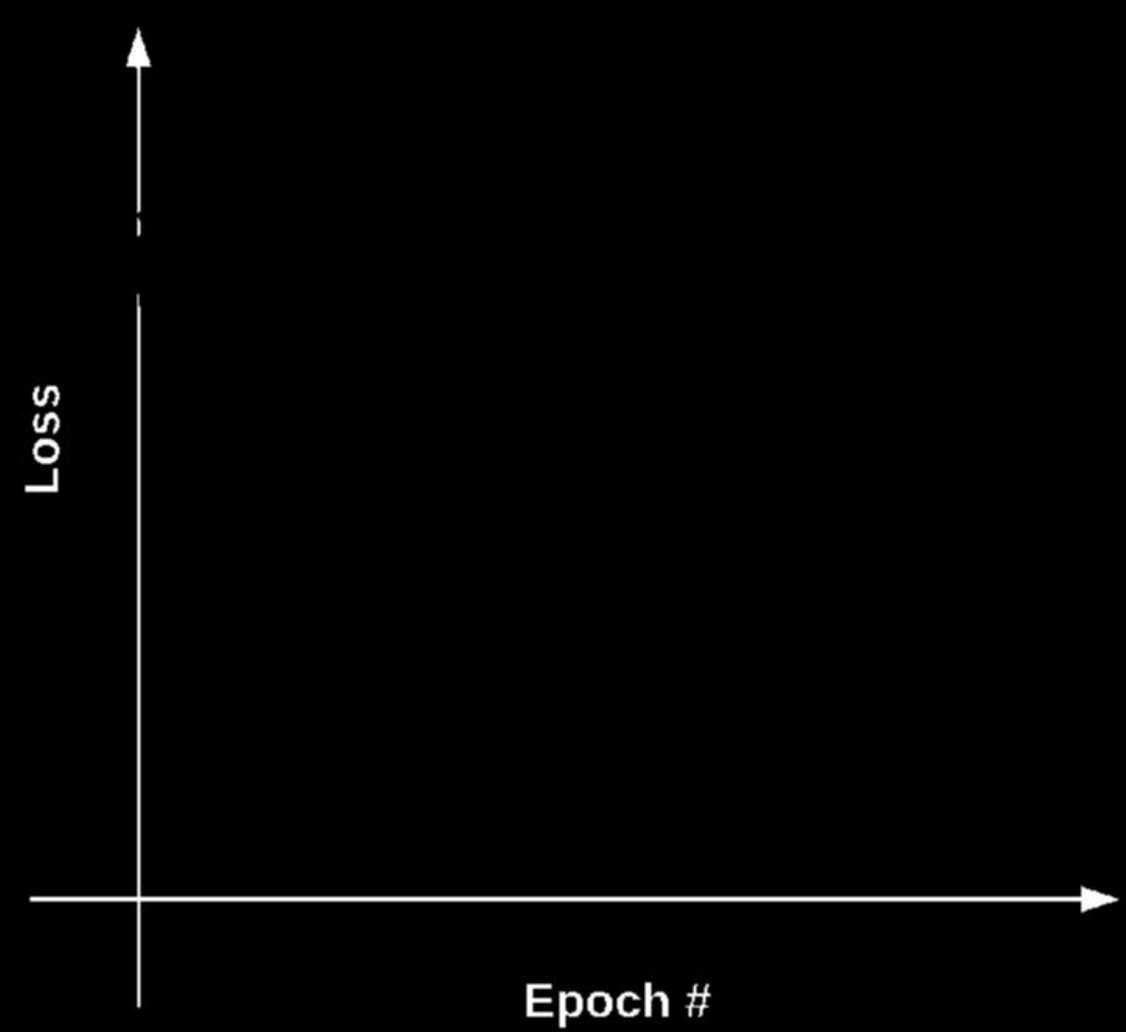

Figure 17.2: A plot visualizing the affect varying learning rates will have on loss (heavily inspired by Karpathy et al. [93]). Very high learning rates (**red**) will drop loss initially but dramatically increase soon after. Low learning rates (**blue**) will be approximately linear in their loss over 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*.

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*.

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.

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.

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
2       keras.callbacks        BaseLogger
3       matplotlib.pyplot    plt
4       numpy    np
5       json
6       os
7
8      TrainingMonitor(BaseLogger):
9                 (    , figPath, jsonPath=    , startAt= ):
10
11
12              (TrainingMonitor,     ).        ()
13             .figPath = figPath
14             .jsonPath = jsonPath
15             .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                          (     , logs={}):
18
19                  .H = {}
20
21
22                     .jsonPath            :
23                      os.path.exists(     .jsonPath):
24                            .H = json.loads(     (     .jsonPath).read())
25
26
27                                .startAt >   :
28
29
30
31                                  k         .H.keys():
32                                        .H[k] =      .H[k][:     .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                          (     , epoch, logs={}):
35
36
37                  (k, v)    logs.items():
38                   l =     .H.get(k, [])
39                   l.append(v)
40                       .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
43
44                     .jsonPath            :
45                  f =     (     .jsonPath, "w")
```

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

Finally, we can construct the actual plot as well:

```
49
50
51                  (    .H["loss"]) >  :
52
53                  N = np.arange( ,    (    .H["loss"]))
54                  plt.style.use("ggplot")
55                  plt.figure()
56                  plt.plot(N,     .H["loss"], label="train_loss")
57                  plt.plot(N,     .H["val_loss"], label="val_loss")
58                  plt.plot(N,     .H["acc"], label="train_acc")
59                  plt.plot(N,     .H["val_acc"], label="val_acc")
60                  plt.title("Training Loss and Accuracy [Epoch {}]".format(
61                          (    .H["loss"])))
62                  plt.xlabel("Epoch #")
63                  plt.ylabel("Loss/Accuracy")
64                  plt.legend()
65
66
67                  plt.savefig(    .figPath)
68                  plt.close()
```

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

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
2         matplotlib
3   matplotlib.use("Agg")
4
5
6       pyimagesearch.callbacks        TrainingMonitor
7       sklearn.preprocessing        LabelBinarizer
8       pyimagesearch.nn.conv        MiniVGGNet
9       keras.optimizers        SGD
10      keras.datasets        cifar10
11        argparse
12        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
15 ap = argparse.ArgumentParser()
16 ap.add_argument(" o", "  output", required=    ,
17      help="path to the output directory")
18 args =     (ap.parse_args())
19
20
21      ("[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
24
25      ("[INFO] loading CIFAR 10 data...")
26 ((trainX, trainY), (testX, testY)) = cifar10.load_data()
27 trainX = trainX.astype("float") /
28 testX = testX.astype("float") /
29
30
31 lb = LabelBinarizer()
32 trainY = lb.fit_transform(trainY)
33 testY = lb.transform(testY)
34
35
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
40      ("[INFO] compiling model...")
41 opt = SGD(lr=    , momentum=   , nesterov=    )
42 model = MiniVGGNet.build(width=  , height=  , depth= , classes=  )
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
47 figPath = os.path.sep.join([args["output"], "{}.png".format(
48      os.getpid())])
```

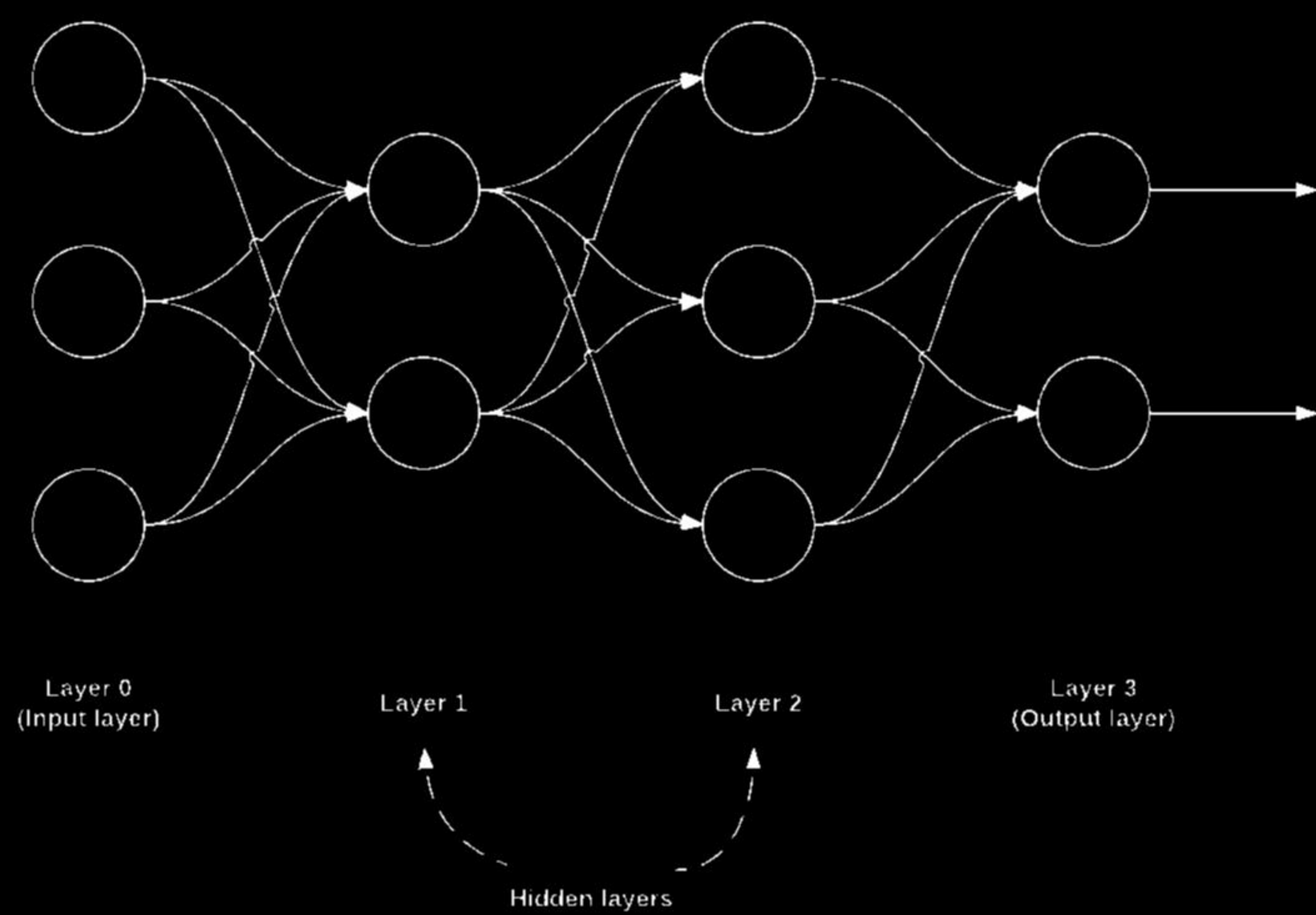

Figure 2.4: A multi-layer, feedforward network architecture with an input layer (3 nodes), two hidden layers (2 nodes in the first layer and 3 nodes in the second layer), and an output layer (2 nodes).

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

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

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

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

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

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

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

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/
```

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

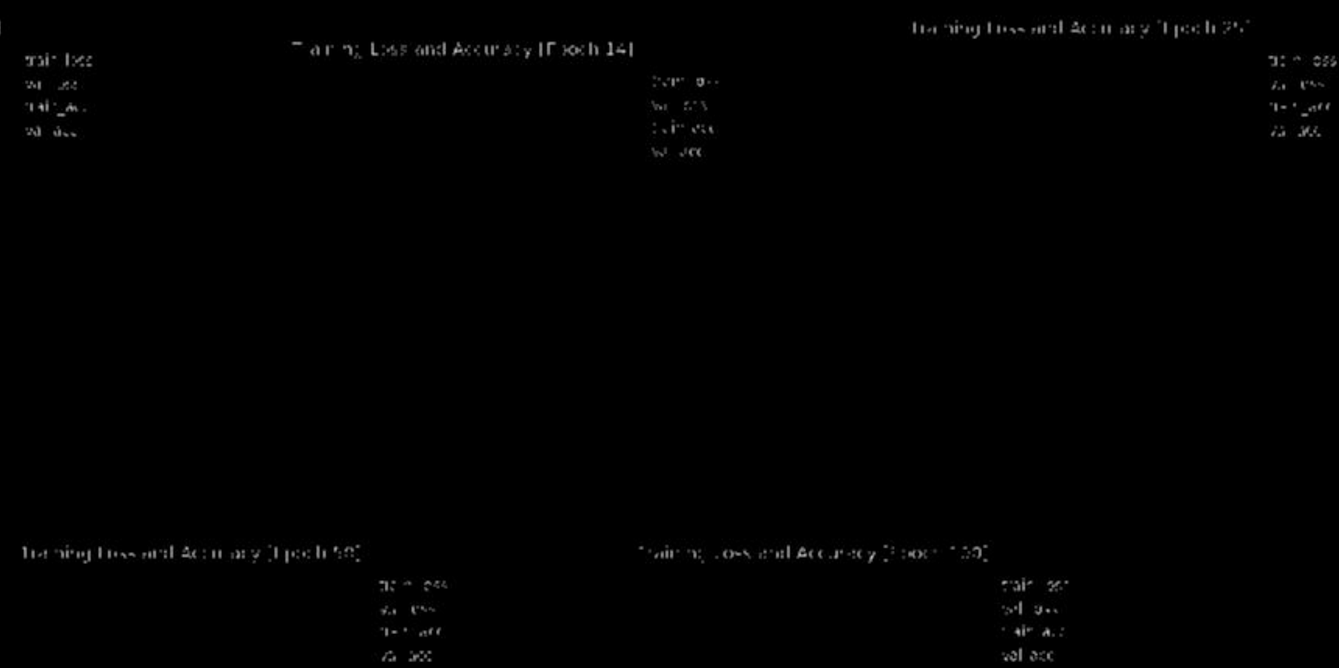

Figure 17.3: Examples of monitoring the training process by examining the training/validation loss curves. At epoch 5 we are still in the underfitting zone. At epoch 14 we are starting to overfit, but nothing to be overly concerned about. By epoch 25 we are certainly inside the overfitting zone. Epochs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

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.

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:

```

sklearn.preprocessing        LabelBinarizer
pyimagesearch.nn.conv        MiniVGGNet
keras.callbacks        ModelCheckpoint
keras.optimizers        SGD
keras.datasets        cifar10
  argparse
  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
11 ap = argparse.ArgumentParser()
12 ap.add_argument(" w", "  weights", required=    ,
13      help="path to weights directory")
14 args =     (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
17
18      ("[INFO] loading CIFAR 10 data...")
19 ((trainX, trainY), (testX, testY)) = cifar10.load_data()
20 trainX = trainX.astype("float") /
21 testX = testX.astype("float") /
22
23
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
29      ("[INFO] compiling model...")
30 opt = SGD(lr=    , decay=    /   , momentum=   , nesterov=    )
31 model = MiniVGGNet.build(width=  , height=  , depth= , classes=  )
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
36
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=    , verbose= )
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
44      ("[INFO] training network...")
45 H = model.fit(trainX, trainY, validation_data=(testX, testY),
46      batch_size=  , epochs=  , callbacks=callbacks, verbose= )
```

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      samples, validate on      samples
Epoch  /40
171s - loss:  .6700 - acc:  .4375 - val_loss:  .2697 - val_acc:  .5425
Epoch  /40
Epoch      : val_loss improved from  .26973 to  .98481, saving model to test/
     weights-001-0.9848.hdf5
...
Epoch   /40
Epoch      : val_loss did not improve
315s - loss:  .2594 - acc:  .9075 - val_loss:  .5707 - val_acc:  .8190
```

•

```
315s - loss: 0.3161 - acc: 0.8867 - val_loss: 0.5704 - val_acc: 0.8175
Epoch 31/40
Epoch 00030: val_loss did not improve
318s - loss: 0.3060 - acc: 0.8899 - val_loss: 0.5649 - val_acc: 0.8188
Epoch 32/40
Epoch 00031: val_loss did not improve
316s - loss: 0.3062 - acc: 0.8907 - val_loss: 0.5640 - val_acc: 0.8207
Epoch 33/40
Epoch 00032: val_loss did not improve
316s - loss: 0.2948 - acc: 0.8947 - val_loss: 0.5738 - val_acc: 0.8158
Epoch 34/40
Epoch 00033: val_loss improved from 0.56279 to 0.55456, saving model to test/weights-033-0.5546.hdf5
316s - loss: 0.2949 - acc: 0.8940 - val_loss: 0.5546 - val_acc: 0.8218
Epoch 35/40
Epoch 00034: val_loss did not improve
314s - loss: 0.2854 - acc: 0.8969 - val_loss: 0.5704 - val_acc: 0.8187
Epoch 36/40
Epoch 00035: val_loss did not improve
316s - loss: 0.2836 - acc: 0.8987 - val_loss: 0.5650 - val_acc: 0.8209
Epoch 37/40
Epoch 00036: val_loss did not improve
315s - loss: 0.2697 - acc: 0.9046 - val_loss: 0.5669 - val_acc: 0.8204
Epoch 38/40
Epoch 00037: val_loss did not improve
316s - loss: 0.2680 - acc: 0.9044 - val_loss: 0.5711 - val_acc: 0.8219
Epoch 39/40
Epoch 00038: val_loss did not improve
316s - loss: 0.2634 - acc: 0.9041 - val_loss: 0.6066 - val_acc: 0.8096
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
```

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).

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:

```

sklearn.preprocessing        LabelBinarizer
pyimagesearch.nn.conv        MiniVGGNet
keras.callbacks        ModelCheckpoint
keras.optimizers        SGD
keras.datasets        cifar10
  argparse
```

Then parse our command line arguments:

```

ap = argparse.ArgumentParser()
ap.add_argument(" w", "  weights", required=    ,
     help="path to best model weights file")
args =     (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:

```


     ("[INFO] loading CIFAR 10 data...")
((trainX, trainY), (testX, testY)) = cifar10.load_data()
trainX = trainX.astype("float") /
testX = testX.astype("float") /


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

As well as initialize our SGD optimizer and MiniVGGNet architecture:

```

     ("[INFO] compiling model...")
opt = SGD(lr=    , decay=     /   , momentum=   , nesterov=    )
model = MiniVGGNet.build(width=  , height=  , depth= , classes=  )
model.compile(loss="categorical_crossentropy", optimizer=opt,
     metrics=["accuracy"])
```

We are now ready to update the `ModelCheckpoint` code:

```


checkpoint = ModelCheckpoint(args["weights"], monitor="val_loss",
     save_best_only=    , verbose= )
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:

```

     ("[INFO] training network...")
H = model.fit(trainX, trainY, validation_data=(testX, testY),
     batch_size=  , epochs=  , callbacks=callbacks, verbose= )
```

To execute our script, issue the following command:

```
python cifar10_checkpoint_best.py --weights test_best/cifar10_best_weights.hdf5
```

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

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

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

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

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

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

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

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

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

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

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/
```

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).

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.

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.

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         == .5.2
pip install pydot-ng== .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.

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
2       pyimagesearch.nn.conv        LeNet
3       keras.utils        plot_model
4
5
6
7  model = LeNet.build(  ,   ,  ,   )
8  plot_model(model, to_file="lenet.png", show_shapes=    )
```

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:

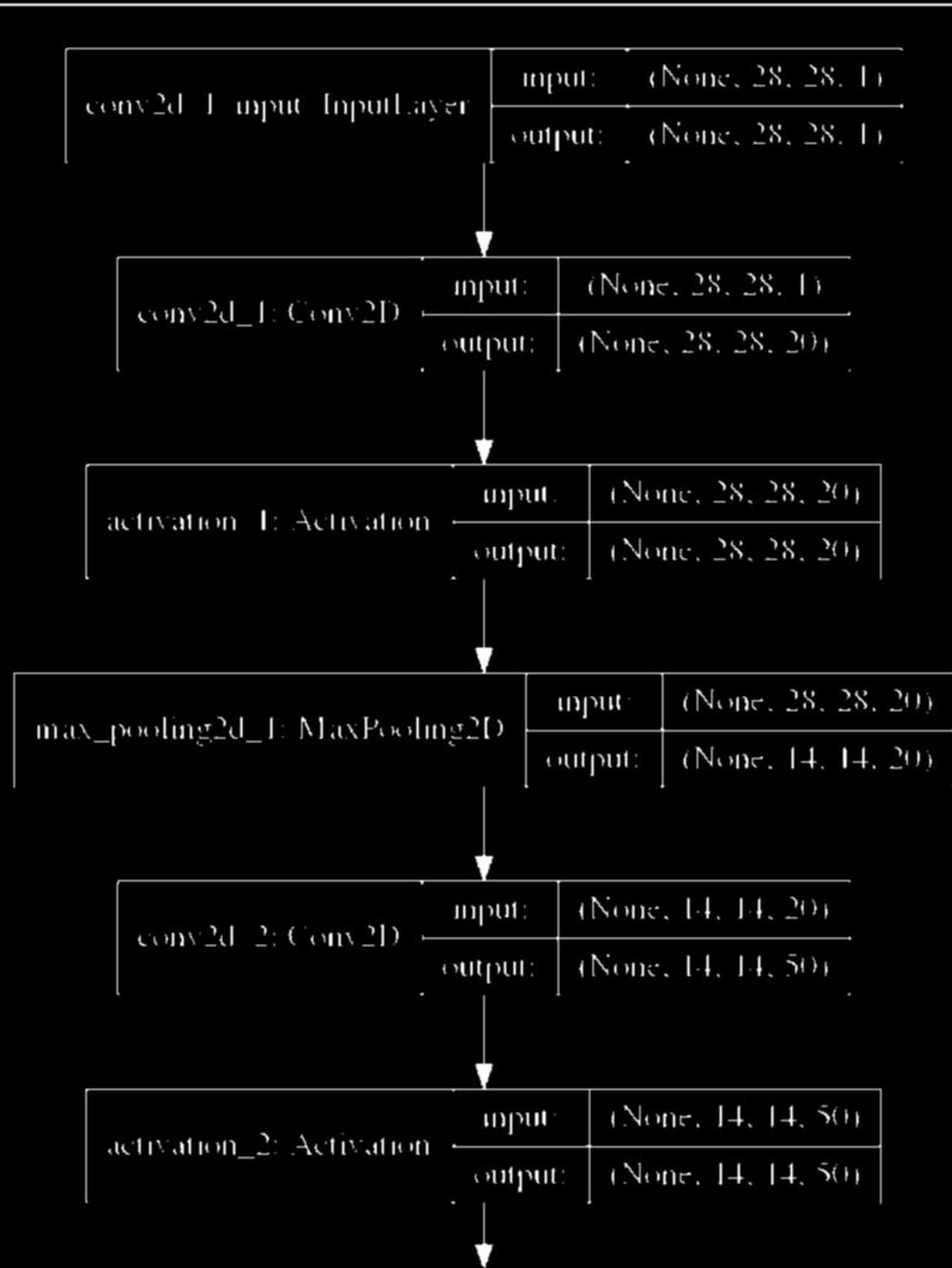

Figure 19.1: Part I of a graphical depiction of the LeNet network architecture generated by Keras. Each node in the graph represents a specific layer function (i.e., convolution, pooling, activation, flattening, fully-connected, etc.). Arrows represent the flow of data through the network. Each node 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
```

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.

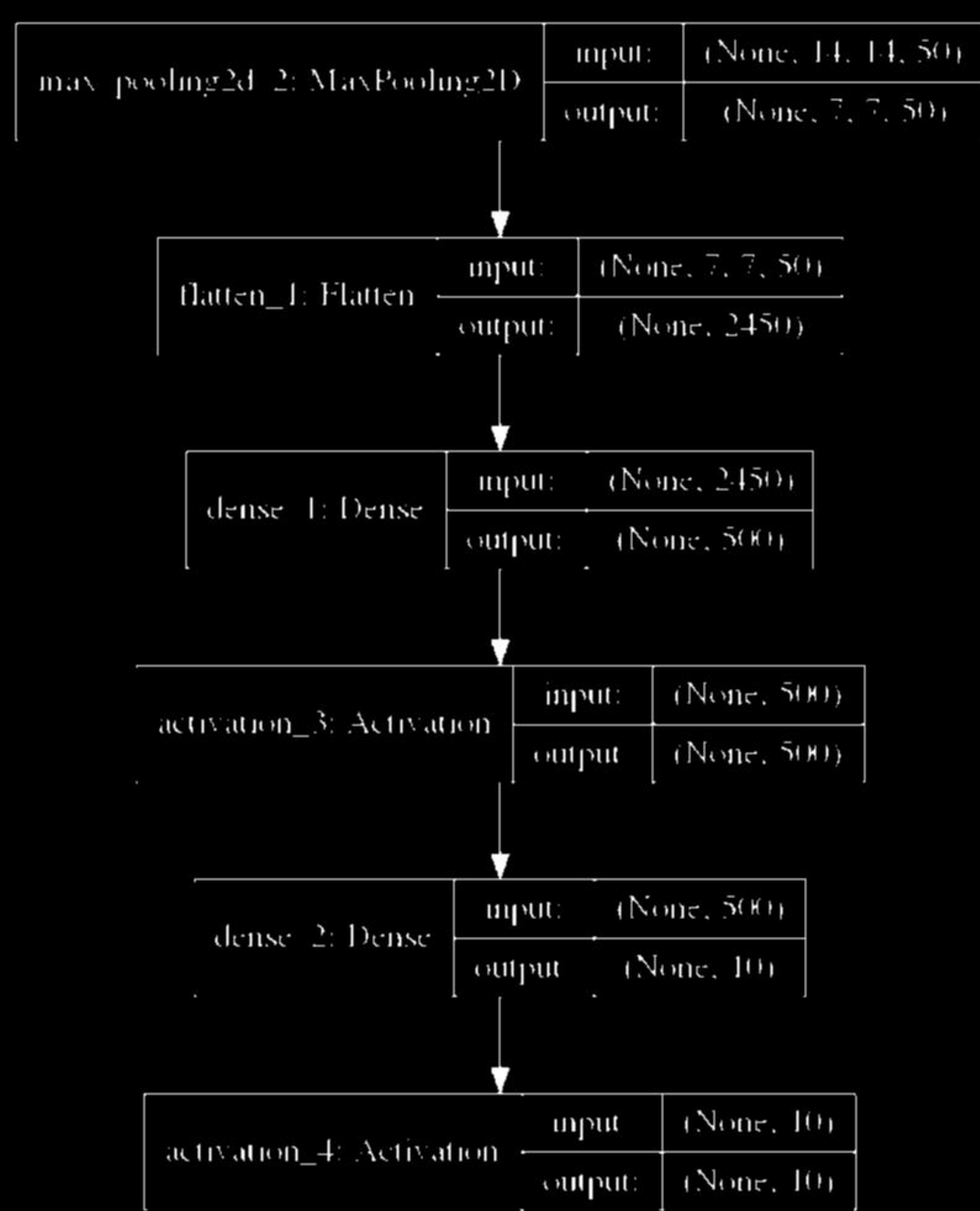

Figure 19.2: Part II of the LeNet architecture visualization, including the fully-connected layers and softmax classifier. In this case, we assume our instantiation of LeNet will be used with the MNIST dataset so we have ten total output nodes in our final softmax layer.

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.

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 .

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.

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.

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:*

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].

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

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

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

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

How "Deep" Is Deep?

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

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

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

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

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.

ResNet

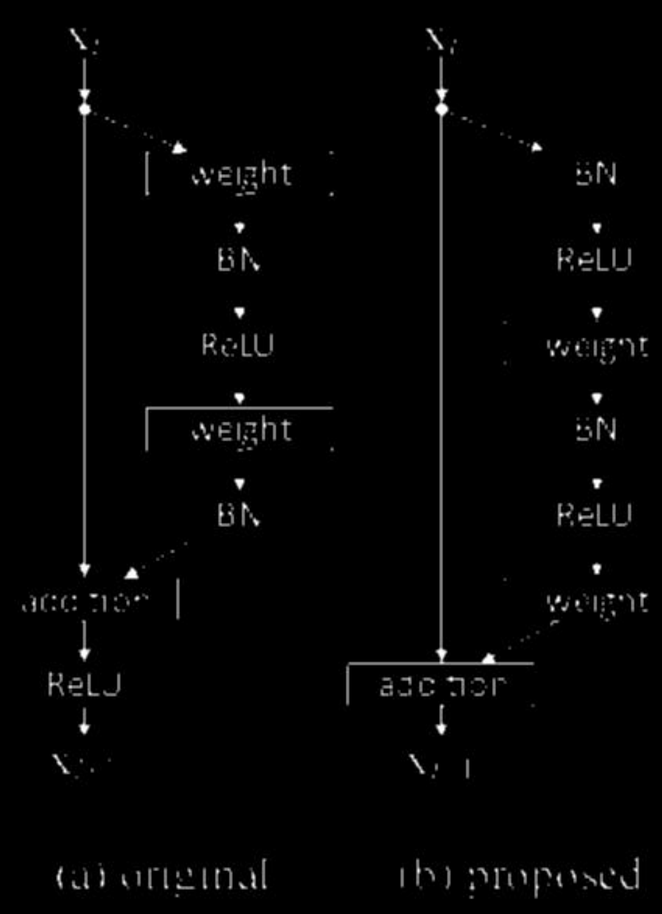

Figure 20.2: **Left:** The original residual module. **Right:** The updated residual module using pre-activation. Figures from He et al., 2016 [130].

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.

Inception V3

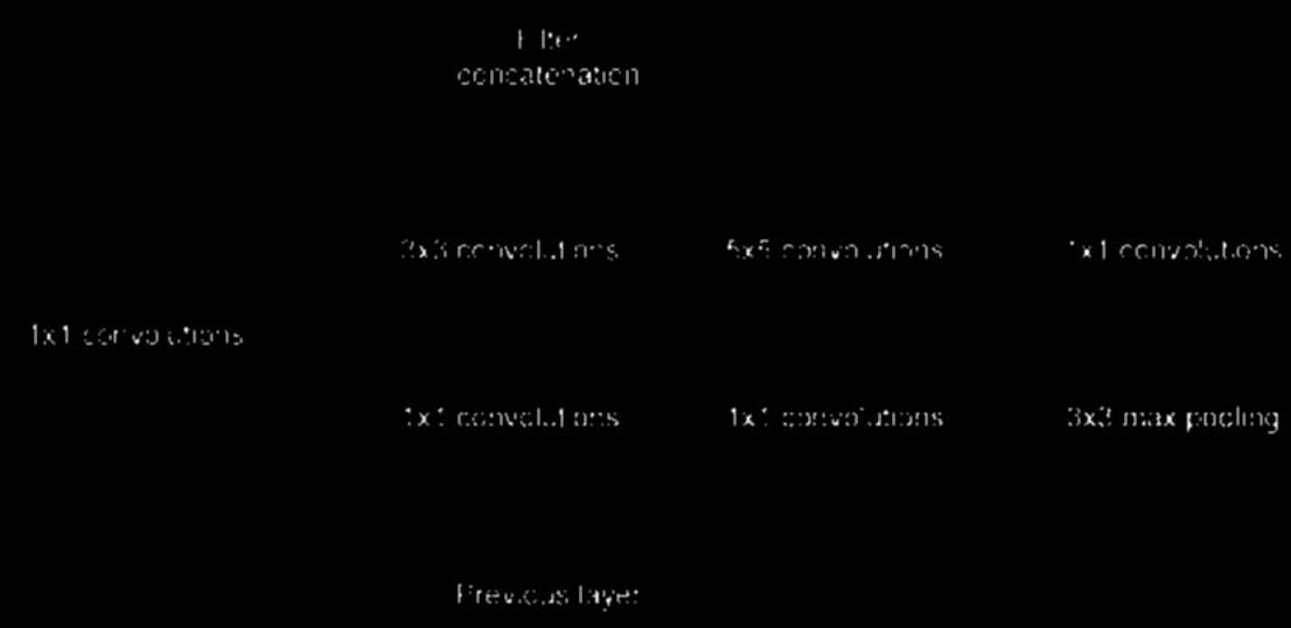

Figure 20.3: The original Inception module used in GoogLeNet. The Inception module acts as a "multi-level feature extractor" by computing 1×1, 3×3, and 5×5 convolutions within the *same* module of the network. Figure from Szegedy et al., 2014 [97].

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*.

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.

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*.

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
2       keras.applications        ResNet50
3       keras.applications        InceptionV3
4       keras.applications        Xception
5       keras.applications        VGG16
6       keras.applications        VGG19
7       keras.applications        imagenet_utils
8       keras.applications.inception_v3        preprocess_input
9       keras.preprocessing.image        img_to_array
10      keras.preprocessing.image        load_img
11        numpy    np
12        argparse
13        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
16  ap = argparse.ArgumentParser()
17  ap.add_argument(" i", "  image", required=    ,
18      help="path to the input image")
19  ap.add_argument(" model", "  model",     =   , default="vgg16",
20      help="name of pre trained network to use")
21  args =     (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
24
25  MODELS = {
26      "vgg16": VGG16,
27      "vgg19": VGG19,
28      "inception": InceptionV3,
29      "xception": Xception,
30      "resnet": ResNet50
31  }
32
33
34     args["model"]        MODELS.keys():
35            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
39
40
41  inputShape = (    ,    )
```

```
42 preprocess = imagenet_utils.preprocess_input
43
44
45
46
47    args["model"]    ("inception", "xception"):
48      inputShape = (    ,    )
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* .

The next step is to load our pre-trained network architecture weights from disk and instantiate our model:

```
51
52
53
54
55
56
57        ("[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
62
63
64        ("[INFO] loading and pre processing image...")
65  image = load_img(args["image"], target_size=inputShape)
66  image = img_to_array(image)
67
68
69
70
71
```

```
72  image = np.expand_dims(image, axis= )
73
74
75
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
79      ("[INFO] classifying image with '{}'...".format(args["model"]))
80  preds = model.predict(image)
81  P = imagenet_utils.decode_predictions(preds)
82
83
84
85      (i, (imagenetID, label, prob))            (P[ ]):
86          ("{}. {}: {:.2f}%".format(i +  , label, prob *   ))
```

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
89
90  orig = cv2.imread(args["image"])
91  (imagenetID, label, prob) = P[ ][ ]
92  cv2.putText(orig, "Label: {}".format(label), (  ,   ),
93      cv2.FONT_HERSHEY_SIMPLEX,    , ( ,    ,  ),  )
94  cv2.imshow("Classification", orig)
95  cv2.waitKey( )
```

To see our pre-trained ImageNet networks in action, let's move on to the next section.

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.

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

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.

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.

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

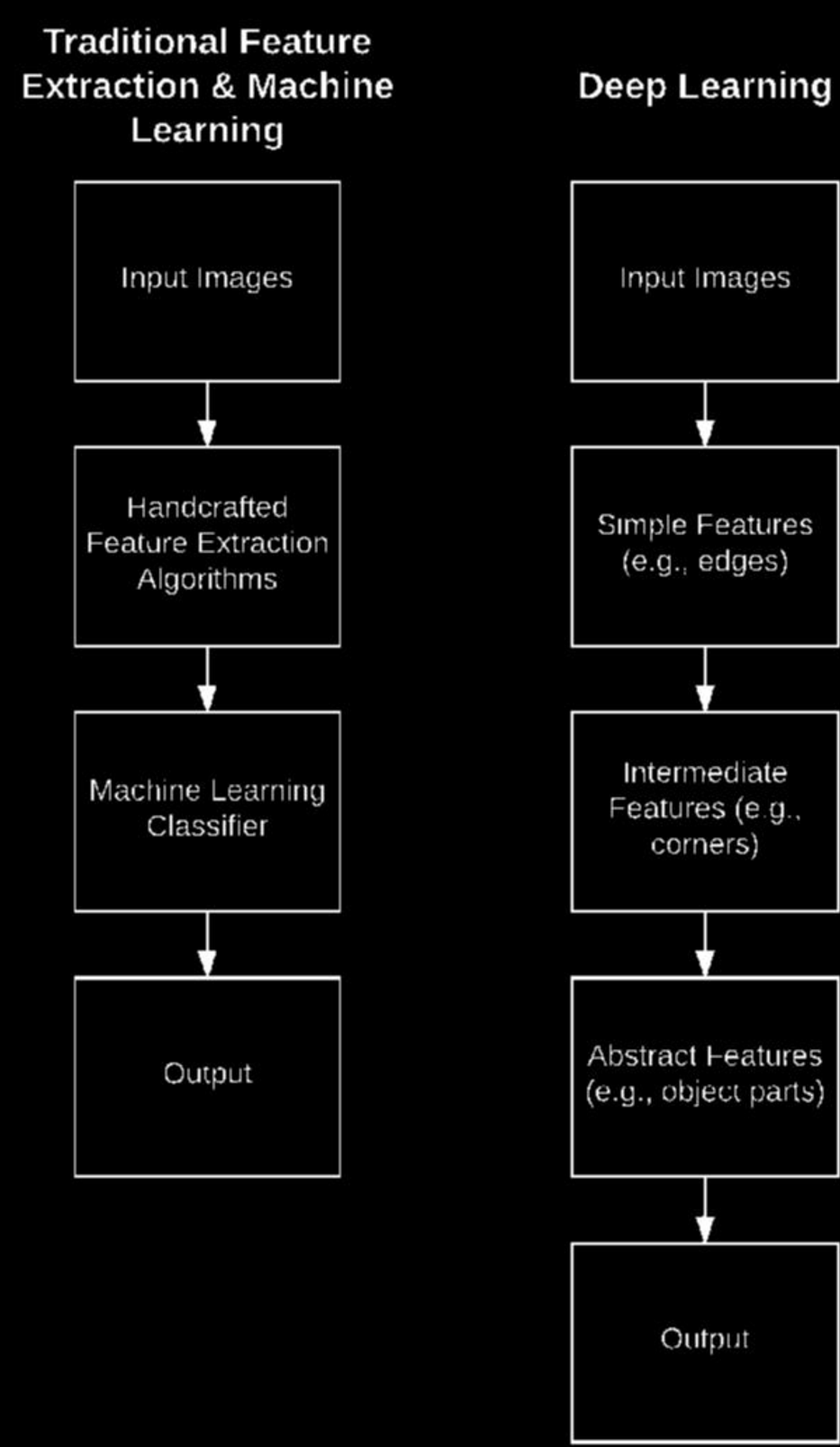

Figure 2.7: **Left:** Traditional process of taking an input set of images, applying hand-designed feature extraction algorithms, followed by training a machine learning classifier on the features. **Right:** Deep learning approach of stacking layers on top of each other that *automatically* learn more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

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

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

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

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

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

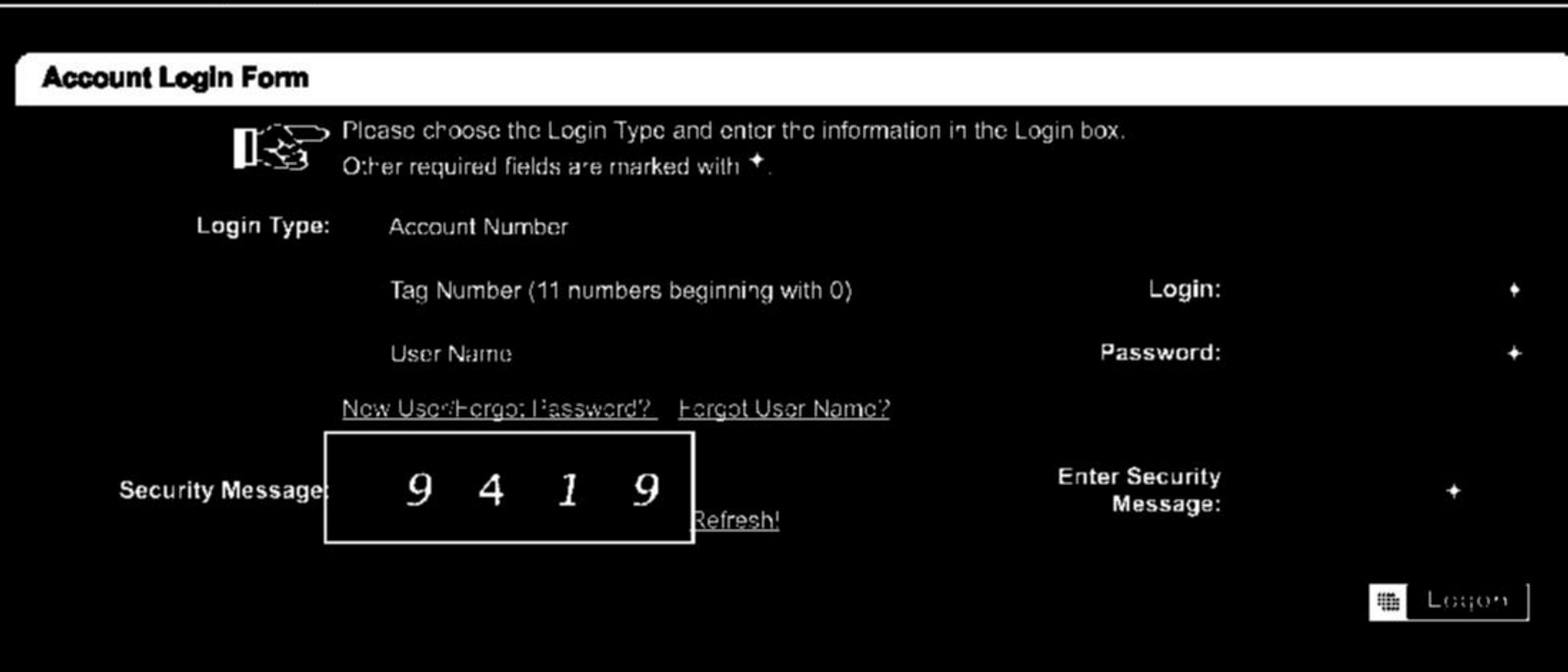

Figure 21.1: The E-Z Pass New York login form. Can you spot the flaw in their login system?

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.

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.

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
2         argparse
3         requests
4         time
5         os
6
7
8  ap = argparse.ArgumentParser()
9  ap.add_argument(" o", "  output", required=    ,
10      help="path to output directory of images")
11 ap.add_argument(" n", "  num images",     =   ,
12      default=   , help="# of images to download")
13 args =     (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
16
17
18 url = "https://www.e zpassny.com/vector/jcaptcha.do"
19 total =
```

We are now ready to download the captcha images:

```
21
22     i         ( , args["num_images"]):
```

```
23          :
24
25              r = requests.get(url, timeout=  )
26
27
28              p = os.path.sep.join([args["output"], "{}.jpg".format(
29                      (total).zfill( ))])
30              f =     (p, "wb")
31              f.write(r.content)
32              f.close()
33
34
35                  ("[INFO] downloaded: {}".format(p))
36          total +=
37
38
39              :
40                  ("[INFO] error downloading image...")
41
42
43      time.sleep(   )
```

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
```

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.

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
2       imutils        paths
3         argparse
4         imutils
5         cv2
6         os
7
8
9  ap = argparse.ArgumentParser()
10 ap.add_argument(" i", "  input", required=    ,
11      help="path to input directory of images")
12 ap.add_argument(" a", "  annot", required=    ,
13      help="path to output directory of annotations")
14 args =     (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
17
18 imagePaths =     (paths.list_images(args["input"]))
19 counts = {}
```

The actual annotation process starts below:

```
21
22     (i, imagePath)            (imagePaths):
23
24           ("[INFO] processing image {}/{}".format(i +  ,
25              (imagePaths)))
26
27         :
28
29
30
31           image = cv2.imread(imagePath)
32           gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
33           gray = cv2.copyMakeBorder(gray,  ,  ,  ,  ,
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,):

```
36
37          thresh = cv2.threshold(gray,  ,    ,
38               cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU)[ ]
```

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
41
42          cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
43               cv2.CHAIN_APPROX_SIMPLE)
44          cnts = cnts[ ]    imutils.is_cv2()      cnts[ ]
45          cnts =      (cnts, key=cv2.contourArea, reverse=    )[: ]
```

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
48              c    cnts:
```

```
49
50
51                  (x, y, w, h) = cv2.boundingRect(c)
52                  roi = gray[y -  :y + h +  , x -  :x + w +  ]
53
54
55
56                  cv2.imshow("ROI", imutils.resize(roi, width=  ))
57                  key = cv2.waitKey( )
```

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.

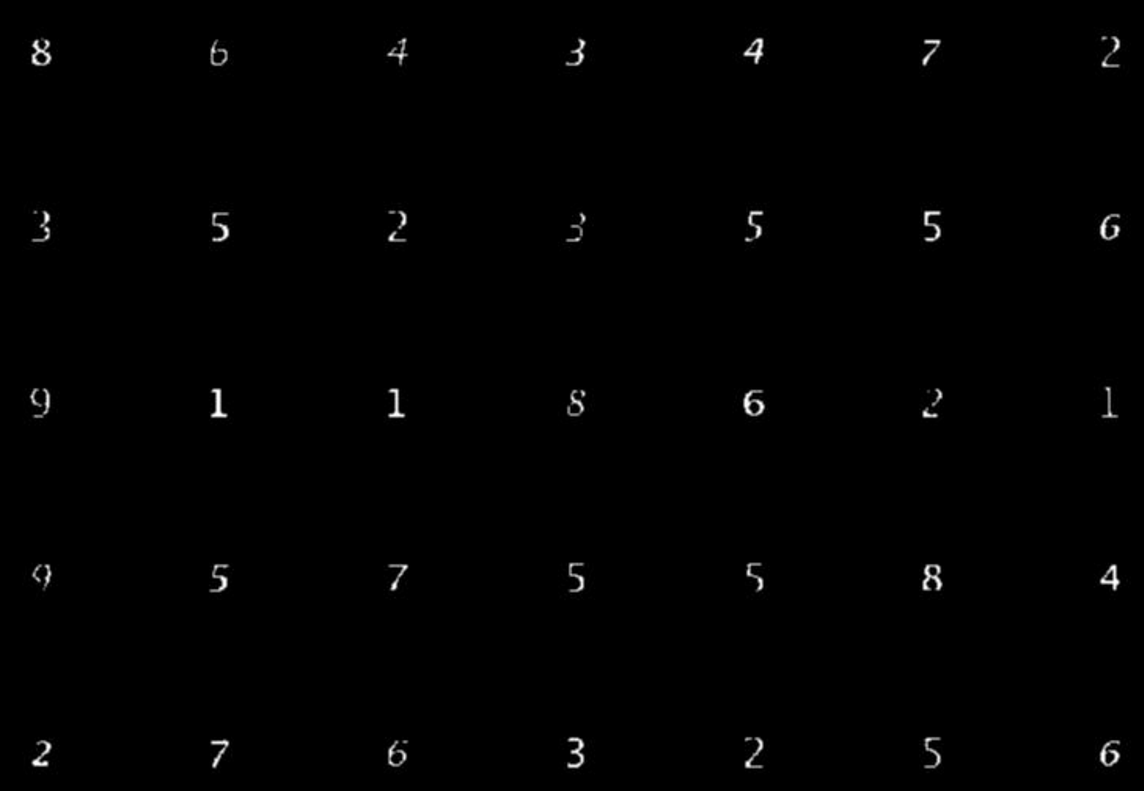

Figure 21.4: A sample of the digit ROIs extracted from our captcha images. Our goal will be to label these images in such a way that we can train a custom Convolutional Neural Network on them.

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
60                  key ==    ("`"):
61                         ("[INFO] ignoring character")
62
63
64
65
66                  key =    (key).upper()
67                  dirPath = os.path.sep.join([args["annot"], key])
68
```

```
69
70                          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
74                  count = counts.get(key,  )
75                  p = os.path.sep.join([dirPath, "{}.png".format(
76                          (count).zfill( ))])
77                  cv2.imwrite(p, roi)
78
79
80                  counts[key] = count +
```

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
83
84
85              KeyboardInterrupt:
86                  ("[INFO] manually leaving script")
87
88
89
90              :
91                  ("[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/

ls -l dataset/1/*.png | wc -l
```

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.

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
2         imutils
3         cv2
4
5                (image, width, height):
6
7
8       (h, w) = image.shape[: ]
9
10
11
12         w > h:
13           image = imutils.resize(image, width=width)
14
15
16
17          :
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
21
22      padW =    ((width - image.shape[ ]) /    )
23      padH =    ((height - image.shape[ ]) /    )
24
25
26
27      image = cv2.copyMakeBorder(image, padH, padH, padW, padW,
28           cv2.BORDER_REPLICATE)
29      image = cv2.resize(image, (width, height))
30
31
32              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.

Copyright © 2017 Adrian Rosebrock, PyImageSearch.com

PUBLISHED BY PYIMAGESEARCH

PYIMAGESEARCH.COM

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

First printing, September 2017

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

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

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

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

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

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

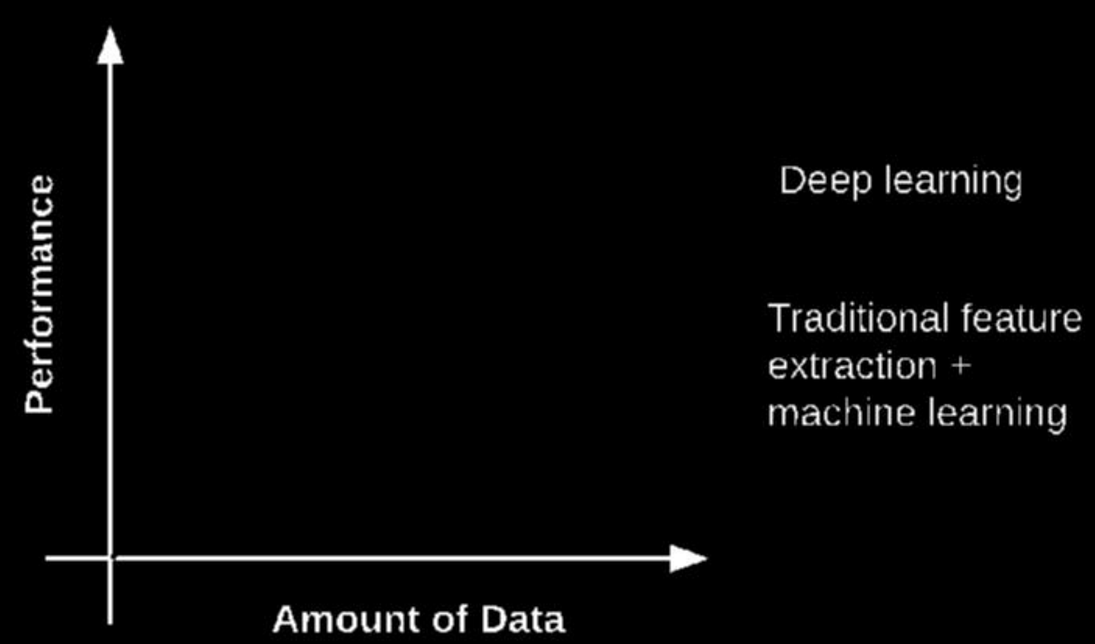

Figure 2.8: As the amount of data available to deep learning algorithms increases, accuracy does as well, substantially outperforming traditional feature extraction + machine learning approaches.

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

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

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

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

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
2       sklearn.preprocessing        LabelBinarizer
3       sklearn.model_selection        train_test_split
4       sklearn.metrics        classification_report
5       keras.preprocessing.image        img_to_array
6       keras.optimizers        SGD
7       pyimagesearch.nn.conv        LeNet
8       pyimagesearch.utils.captchahelper        preprocess
9       imutils        paths
10        matplotlib.pyplot    plt
11        numpy    np
12        argparse
13        cv2
14        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
17  ap = argparse.ArgumentParser()
18  ap.add_argument(" d", "  dataset", required=    ,
19       help="path to input dataset")
20  ap.add_argument(" m", "  model", required=    ,
21       help="path to output model")
22  args =     (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
25  data = []
26  labels = []
27
28
29      imagePath    paths.list_images(args["dataset"]):
30
31       image = cv2.imread(imagePath)
32       image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
33       image = preprocess(image,    ,    )
34       image = img_to_array(image)
35       data.append(image)
```

```
36
37
38
39      label = imagePath.split(os.path.sep)[- ]
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
43  data = np.array(data, dtype="float") /
44  labels = np.array(labels)
45
46
47
48  (trainX, testX, trainY, testY) = train_test_split(data,
49       labels, test_size=    , random_state=  )
50
51
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
57       ("[INFO] compiling model...")
58  model = LeNet.build(width=  , height=  , depth= , classes= )
59  opt = SGD(lr=    )
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:

```

     ("[INFO] training network...")
H = model.fit(trainX, trainY,  validation_data=(testX, testY),
     batch_size=  , epochs=  , verbose= )


     ("[INFO] evaluating network...")
predictions = model.predict(testX, batch_size=  )
     (classification_report(testY.argmax(axis= ),
     predictions.argmax(axis= ), target_names=lb.classes_))


     ("[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:

```

plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange( ,   ), H.history["loss"], label="train_loss")
plt.plot(np.arange( ,   ), H.history["val_loss"], label="val_loss")
plt.plot(np.arange( ,   ), H.history["acc"], label="acc")
plt.plot(np.arange( ,   ), 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
```

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.

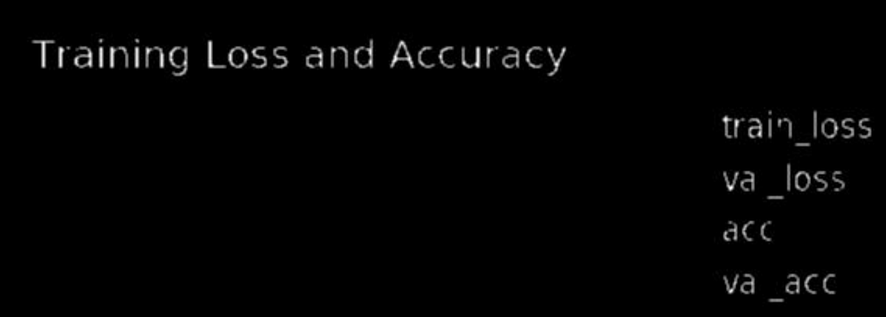

Figure 21.6: Using the LeNet architecture on our custom digits datasets enables us to obtain 100% classification accuracy after only fifteen epochs. Furthermore, there are no signs of overfitting.

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

```
ls -l output/
```

We can then use this model on new input images.

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:

```

keras.preprocessing.image        img_to_array
keras.models        load_model
pyimagesearch.utils.captchahelper        preprocess
imutils        contours
imutils        paths
  numpy    np
  argparse
  imutils
  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:

```

ap = argparse.ArgumentParser()
ap.add_argument(" i", "  input", required=    ,
     help="path to input directory of images")
ap.add_argument(" m", "  model", required=    ,
     help="path to input model")
args =     (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:

```

     ("[INFO] loading pre trained network...")
model = load_model(args["model"])


imagePaths =     (paths.list_images(args["input"]))
imagePaths = np.random.choice(imagePaths, size=(  ,),
     replace=     )
```

Here comes the fun part – actually breaking the captcha:

```

    imagePath    imagePaths:



     image = cv2.imread(imagePath)
```

```
35      gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
36      gray = cv2.copyMakeBorder(gray,    ,    ,    ,    ,
37           cv2.BORDER_REPLICATE)
38
39
40      thresh = cv2.threshold(gray,   ,     ,
41           cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU)[ ]
```

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
44
45      cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
46           cv2.CHAIN_APPROX_SIMPLE)
47      cnts = cnts[ ]    imutils.is_cv2()      cnts[ ]
48      cnts =       (cnts, key=cv2.contourArea, reverse=    )[: ]
49      cnts = contours.sort_contours(cnts)[ ]
50
51
52
53      output = cv2.merge([gray] *  )
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 ( ).

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
57          c    cnts:
58
59
60           (x, y, w, h) = cv2.boundingRect(c)
61           roi = gray[y -  :y + h +  , x -  :x + w +  ]
62
63
```

```
64            roi = preprocess(roi,    ,    )
65            roi = np.expand_dims(img_to_array(roi), axis= ) /
66            pred = model.predict(roi).argmax(axis= )[ ] +
67            predictions.append(    (pred))
68
69
70            cv2.rectangle(output, (x -   , y -   ),
71                 (x + w +   , y + h +   ), ( ,     ,   ),   )
72            cv2.putText(output,     (pred), (x -   , y -   ),
73                 cv2.FONT_HERSHEY_SIMPLEX,      , ( ,     ,   ),   )
```

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
76        ("[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
```

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.

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.

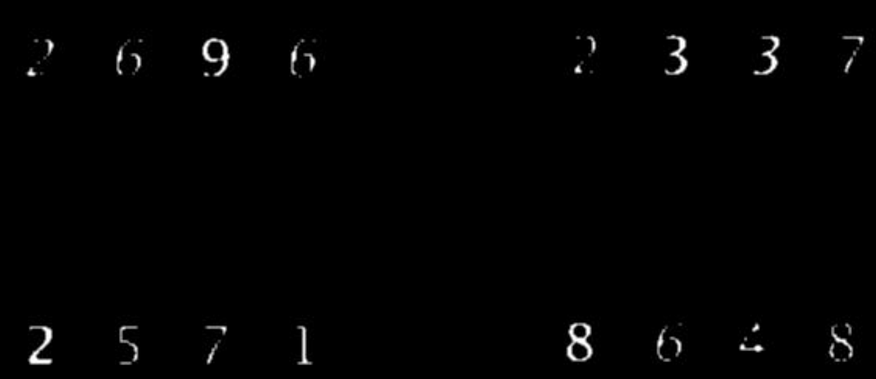

Figure 21.7: Examples of captchas that have been correctly classified and broken by our LeNet model.

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.
[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 – 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 . 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.

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.

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:

```

sklearn.preprocessing        LabelEncoder
sklearn.model_selection        train_test_split
sklearn.metrics        classification_report
keras.preprocessing.image        img_to_array
keras.utils        np_utils
pyimagesearch.nn.conv        LeNet
imutils        paths
  matplotlib.pyplot    plt
  numpy    np
  argparse
  imutils
  cv2
  os
```

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

Summary

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

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

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

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

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

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

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
17  ap = argparse.ArgumentParser()
18  ap.add_argument(" d", "  dataset", required=    ,
19       help="path to input dataset of faces")
20  ap.add_argument(" m", "  model", required=    ,
21       help="path to output model")
22  args =     (ap.parse_args())
23
24
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
29      imagePath          (    (paths.list_images(args["dataset"]))):
30
31       image = cv2.imread(imagePath)
32       image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
33       image = imutils.resize(image, width=  )
34       image = img_to_array(image)
35       data.append(image)
36
37
38
39       label = imagePath.split(os.path.sep)[- ]
40       label = "smiling"    label == "positives"      "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
44  data = np.array(data, dtype="float") /
45  labels = np.array(labels)
46
47
48  le = LabelEncoder().fit(labels)
49  labels = np_utils.to_categorical(le.transform(labels),  )
```

Our next code block handles our data imbalance issue by computing the class weights:

```
51
52  classTotals = labels.sum(axis= )
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
56
57  (trainX, testX, trainY, testY) = train_test_split(data,
58       labels, test_size=    , stratify=labels, random_state=  )
```

Finally, we are ready to train LeNet:

```
60
61       ("[INFO] compiling model...")
62  model = LeNet.build(width=  , height=  , depth= , classes= )
63  model.compile(loss="binary_crossentropy", optimizer="adam",
64       metrics=["accuracy"])
65
66
67       ("[INFO] training network...")
68  H = model.fit(trainX, trainY, validation_data=(testX, testY),
69       class_weight=classWeight, batch_size=  , epochs=  , verbose= )
```

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
72      ("[INFO] evaluating network...")
73  predictions = model.predict(testX, batch_size=  )
74      (classification_report(testY.argmax(axis= ),
75      predictions.argmax(axis= ), target_names=le.classes_))
76
77
78      ("[INFO] serializing network...")
79  model.save(args["model"])
```

We'll also construct a learning curve for our network so we can visualize performance:

```
81
82  plt.style.use("ggplot")
83  plt.figure()
84  plt.plot(np.arange( ,   ), H.history["loss"], label="train_loss")
85  plt.plot(np.arange( ,   ), H.history["val_loss"], label="val_loss")
86  plt.plot(np.arange( ,   ), H.history["acc"], label="acc")
87  plt.plot(np.arange( ,   ), 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
```

After 15 epochs we can see that our network is obtaining 93% classification accuracy. Figure 22.2 plots our learning curve:

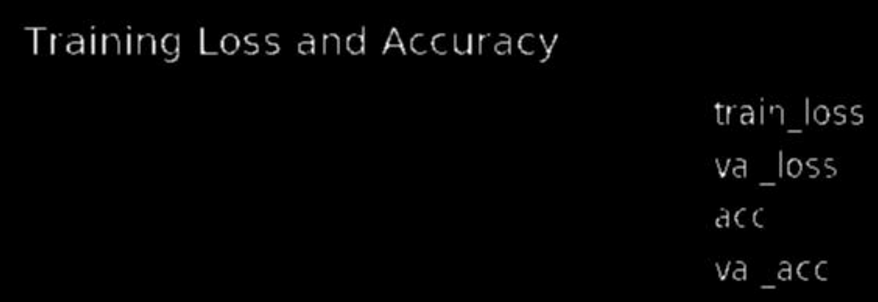

Figure 22.2: A plot of the learning curve for the LeNet architecture trained on the SMILES dataset. After fifteen epochs we are obtaining $\approx 93\%$ classification accuracy on our testing set.

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*.

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
2       keras.preprocessing.image        img_to_array
3       keras.models        load_model
4       numpy    np
5       argparse
6       imutils
7       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
10  ap = argparse.ArgumentParser()
11  ap.add_argument(" c", "  cascade", required=    ,
12       help="path to where the face cascade resides")
13  ap.add_argument(" m", "  model", required=    ,
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 =     (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 ( ) along with the Object Detection module inside [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
20  detector = cv2.CascadeClassifier(args["cascade"])
```

```
21 model = load_model(args["model"])
22
23
24         args.get("video",      ):
25       camera = cv2.VideoCapture( )
26
27
28      :
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
32            :
33
34       (grabbed, frame) = camera.read()
35
36
37
38          args.get("video")         grabbed:
39
40
41
42
43       frame = imutils.resize(frame, width=   )
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
48
49       rects = detector.detectMultiScale(gray, scaleFactor=   ,
50            minNeighbors= , minSize=(   ,   ),
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 ( ) 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

.

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
54          (fX, fY, fW, fH)    rects:
55
56
57
58          roi = gray[fY:fY + fH, fX:fX + fW]
59          roi = cv2.resize(roi, (  ,   ))
60          roi = roi.astype("float") /
61          roi = img_to_array(roi)
62          roi = np.expand_dims(roi, axis= )
```

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
65
66          (notSmiling, smiling) = model.predict(roi)[ ]
67          label = "Smiling"    smiling > notSmiling      "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
70
71          cv2.putText(frameClone, label, (fX, fY -   ),
72               cv2.FONT_HERSHEY_SIMPLEX,      , ( ,  ,    ),  )
73          cv2.rectangle(frameClone, (fX, fY), (fX + fW, fY + fH),
74               ( ,  ,    ),  )
```

Our final code block handles displaying the output frame to our screen:

```
76
77      cv2.imshow("Face", frameClone)
78
79
80         cv2.waitKey( ) &      ==    ("q"):
81
82
83
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:

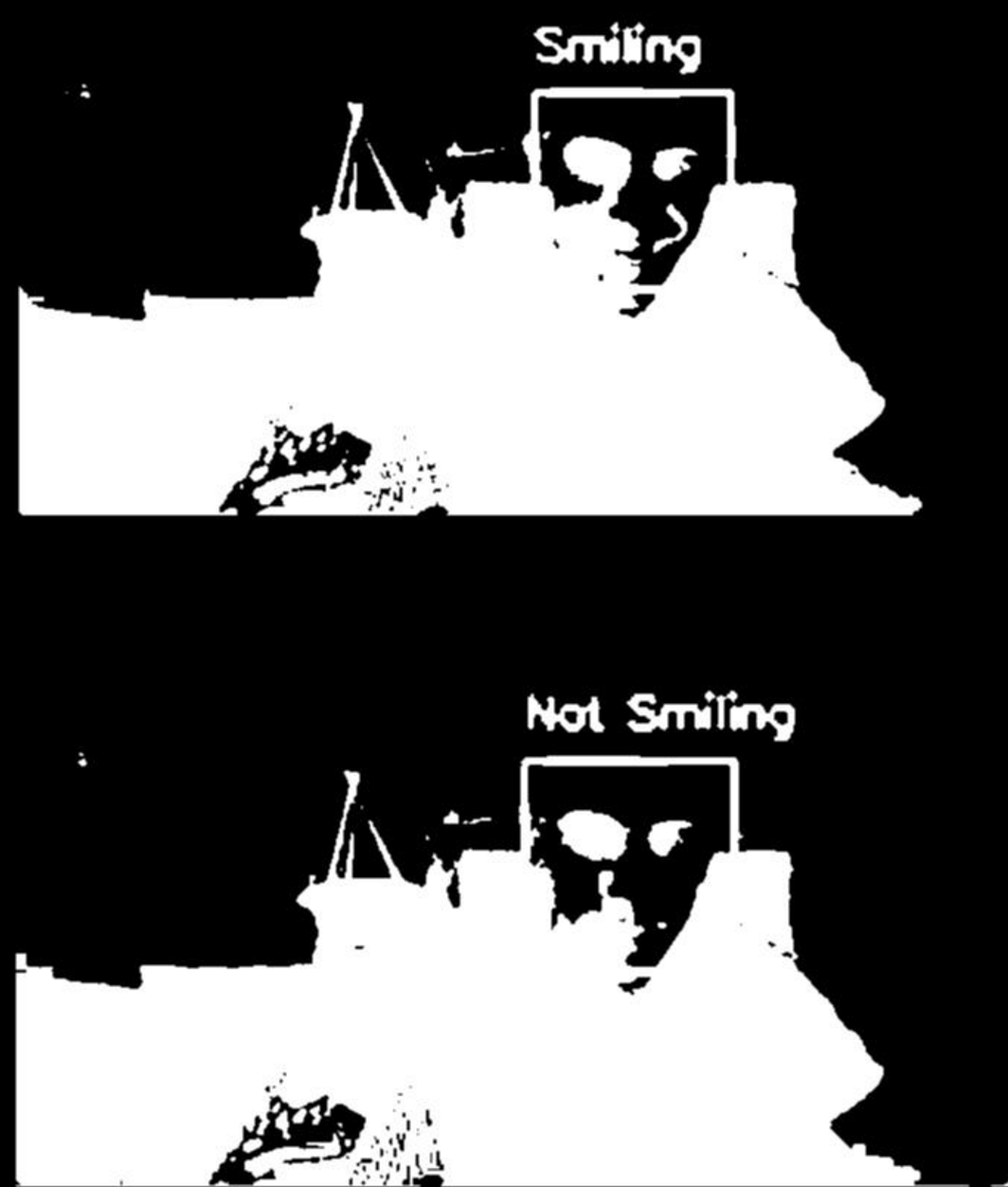

Figure 22.3: Applying our CNN to recognize smiling vs. not-smiling in real-time video streams on a CPU.

Notice how LeNet is correctly predicting "smiling" or "not smiling" based on my facial expression.

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.

3. Image Fundamentals

Before we can start building our own image classifiers, we first need to understand what an image is. We'll start with the buildings blocks of an image – the pixel.

We'll discuss exactly what a pixel is, how they are used to form an image, and how to access pixels that are represented as NumPy arrays (as nearly all image processing libraries in Python do, including OpenCV and scikit-image).

The chapter will conclude with a discussion on the aspect ratio of an image and the relation it has when preparing our image dataset for training a neural network.

Pixels: The Building Blocks of Images

Pixels are the raw building blocks of an image. Every image consists of a set of pixels. There is no finer granularity than the pixel.

Normally, a pixel is considered the "color" or the "intensity" of light that appears in a given place in our image. If we think of an image as a grid, each square contains a single pixel. For example, take a look at Figure 3.1.

The image in Figure 3.1 above has a resolution of $1,000 \times 750$, meaning that it is 1,000 pixels wide and 750 pixels tall. We can conceptualize an image as a (multidimensional) matrix. In this case, our matrix has 1,000 columns (the width) with 750 rows (the height). Overall, there are $1,000 \times 750 = 750,000$ total pixels in our image.

Most pixels are represented in two ways:

1. Grayscale/single channel
2. Color

In a grayscale image, each pixel is a scalar value between 0 and 255, where zero corresponds to "black" and 255 being "white". Values between 0 and 255 are varying shades of gray, where values closer to 0 are darker and values closer to 255 are lighter. The grayscale gradient image in Figure 3.2 demonstrates *darker pixels* on the left-hand side and progressively *lighter pixels* on the right-hand side.

Color pixels; however, are normally represented in the RGB color space (other color spaces do exist, but are outside the scope of this book and not relevant for deep learning).

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...

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:

And if you have any questions at all, feel free to contact me:

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).

[66] Marius Muja and David G. Lowe. "Scalable Nearest Neighbor Algorithms for High Dimensional Data". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 36 (2014) (cited on pages 79, 81).

[67] Sanjoy Dasgupta. "Experiments with Random Projection". In: *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*. UAI '00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pages 143–151. ISBN: 1-55860-709-9. URL: http://dl.acm.org/citation.cfm?id=647234.719759 (cited on page 79).

[68] Ella Bingham and Heikki Mannila. "Random Projection in Dimensionality Reduction: Applications to Image and Text Data". In: *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '01. San Francisco, California: ACM, 2001, pages 245–250. ISBN: 1-58113-391-X. DOI: 10.1145/502512.502546. URL: http://doi.acm.org/10.1145/502512.502546 (cited on page 79).

[69] Sanjoy Dasgupta and Anupam Gupta. "An Elementary Proof of a Theorem of Johnson and Lindenstrauss". In: *Random Struct. Algorithms* 22.1 (Jan. 2003), pages 60–65. ISSN: 1042-9832. DOI: 10.1002/rsa.10073. URL: http://dx.doi.org/10.1002/rsa.10073 (cited on page 79).

[70] Pedro Domingos. "A Few Useful Things to Know About Machine Learning". In: *Commun. ACM* 55.10 (Oct. 2012), pages 78–87. ISSN: 0001-0782. DOI: 10.1145/2347736.2347755. URL: http://doi.acm.org/10.1145/2347736.2347755 (cited on pages 79, 80).

[71] David Mount and Sunil Ayra. *ANN: A Library for Approximate Nearest Neighbor Searching*. https://www.cs.umd.edu/~mount/ANN/. 2010 (cited on page 81).

[72] Erik Bernhardsson. *Annoy: Approximate Nearest Neighbors in C++/Python optimized for memory usage and loading/saving to disk*. https://github.com/spotify/annoy. 2015 (cited on page 81).

[73] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009. ISBN: 0136042597, 9780136042594 (cited on page 82).

[74] Andrej Karpathy. *Linear Classification*. http://cs231n.github.io/linear-classify/ (cited on pages 82, 94).

[75] P.N. Klein. *Coding the Matrix: Linear Algebra Through Applications to Computer Science*. Newtonian Press. 2013. ISBN: 9780615880990. URL: https://books.google.com/books?id=3AA4nwEACAAJ (cited on page 84).

[76] Andrew Ng. *Machine Learning*. https://www.coursera.org/learn/machine-learning (cited on pages 88, 132, 137, 141).

[77] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 0123748569, 9780123748560 (cited on page 88).

[78] Peter Harrington. *Machine Learning in Action*. Greenwich, CT, USA: Manning Publications Co., 2012. ISBN: 1617290181, 9781617290183 (cited on page 88).

[79] Stephen Marsland. *Machine Learning: An Algorithmic Perspective*. 1st. Chapman & Hall/CRC, 2009. ISBN: 1420067184, 9781420067187 (cited on page 88).

[80] Michael Zibulevsky. *Homework on analytical and numerical computation of gradient and Hessian*. https://www.youtube.com/watch?v=ruuW4-InUxM (cited on page 98).

[81] Andrew Ng. *CS229 Lecture Notes*. http://cs229.stanford.edu/notes/cs229-notes1.pdf (cited on page 98).

[82] Andrej Karpathy. *Optimization*. http://cs231n.github.io/optimization-1/ (cited on page 98).

[83] Andrej Karpathy. *Lecture 3: Loss Functions and Optimization*. http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture3.pdf (cited on page 99).

[84] Dmytro Mishkin and Jiri Matas. "All you need is a good init". In: *CoRR* abs/1511.06422 (2015). URL: http://arxiv.org/abs/1511.06422 (cited on page 102).

[85] Stanford University. Stanford Electronics Laboratories et al. *Adaptive "adaline" neuron using chemical "memistors."*. 1960. URL: https://books.google.com/books?id=Yc4EAAAAIAAJ (cited on page 106).

[86] Ning Qian. "On the Momentum Term in Gradient Descent Learning Algorithms". In: *Neural Netw.* 12.1 (Jan. 1999), pages 145–151. ISSN: 0893-6080. DOI: 10.1016/S0893-6080(98)00116-6. URL: http://dx.doi.org/10.1016/S0893-6080(98)00116-6 (cited on pages 111, 112).

[87] Yurii Nesterov. "A method of solving a convex programming problem with convergence rate O (1/k2)". In: *Soviet Mathematics Doklady*. Volume 27. 2. 1983, pages 372–376 (cited on pages 111, 112).

[88] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *CoRR* abs/1609.04747 (2016). URL: http://arxiv.org/abs/1609.04747 (cited on page 112).

[89] Richard S. Sutton. "Two Problems with Backpropagation and Other Steepest-Descent Learning Procedures for Networks". In: *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*. Hillsdale, NJ: Erlbaum, 1986 (cited on page 112).

[90] Geoffrey Hinton. *Neural Networks for Machine Learning*. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf (cited on pages 112, 164).

[91] Yoshua Bengio, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. "Advances in Optimizing Recurrent Networks". In: *CoRR* abs/1212.0901 (2012). URL: http://arxiv.org/abs/1212.0901 (cited on page 112).

[92] Ilya Sutskever. "Training Recurrent Neural Networks". AAINS22066. PhD thesis. Toronto, Ont., Canada, Canada, 2013. ISBN: 978-0-499-22066-0 (cited on page 112).

[93] Andrej Karpathy. *Neural Networks (Part III)*. http://cs231n.github.io/neural-networks-3/ (cited on pages 113, 253).

[94] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Edited by F. Pereira et al. Curran Associates, Inc., 2012, pages 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf (cited on pages 113, 185, 192, 229).

[95] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *CoRR* abs/1409.1556 (2014). URL: http://arxiv.org/abs/1409.1556 (cited on pages 113, 192, 195, 227, 229, 278).

[96] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). URL: http://arxiv.org/abs/1512.03385 (cited on pages 113, 126, 188, 192, 279).

[97] Christian Szegedy et al. "Going Deeper with Convolutions". In: *Computer Vision and Pattern Recognition (CVPR)*. 2015. URL: http://arxiv.org/abs/1409.4842 (cited on pages 113, 192, 280).

[98] Hui Zou and Trevor Hastie. "Regularization and variable selection via the Elastic Net". In: *Journal of the Royal Statistical Society, Series B* 67 (2005), pages 301–320 (cited on pages 113, 116).

[99] DeepLearning.net Contributors. *Deep Learning Documentation: Regularization*. http://deeplearning.net/tutorial/gettingstarted.html#regularization (cited on page 117).

[100] Andrej Karpathy. *Neural Networks (Part II)*. http://cs231n.github.io/neural-networks-2/ (cited on page 117).

[101] Richard HR Hahnloser et al. "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit". In: *Nature* 405.6789 (2000), page 947 (cited on pages 126, 128).

[102] Richard H. R. Hahnloser, H. Sebastian Seung, and Jean-Jacques Slotine. "Permitted and Forbidden Sets in Symmetric Threshold-linear Networks". In: *Neural Comput.* 15.3 (Mar. 2003), pages 621–638. ISSN: 0899-7667. DOI: 10.1162/089976603321192103. URL: http://dx.doi.org/10.1162/089976603321192103 (cited on page 126).

[103] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. "Rectifier nonlinearities improve neural network acoustic models". In: *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*. 2013 (cited on page 126).

[104] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)". In: *CoRR* abs/1511.07289 (2015). URL: http://arxiv.org/abs/1511.07289 (cited on page 126).

[105] Donald Hebb. *The organization of behavior: A neuropsychological theory*. Wiley, 1949 (cited on page 128).

[106] Kishan Mehrotra, Chilukuri K. Mohan, and Sanjay Ranka. *Elements of Artificial Neural Networks*. Cambridge, MA, USA: MIT Press, 1997. ISBN: 0-262-13328-8 (cited on pages 128, 132).

[107] Mikel Olazaran. "A Sociological Study of the Official History of the Perceptrons Controversy". In: *Social Studies of Science* 26.3 (1996), pages 611–659. ISSN: 03063127. URL: http://www.jstor.org/stable/285702 (cited on page 129).

[108] Michael Nielsen. *Chapter 2: How the backpropagation algorithm works*. http://neuralnetworksanddeeple com/chap2.html. 2017 (cited on pages 137, 141).

[109] Matt Mazur. *A Step by Step Backpropagation Example*. https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/. 2015 (cited on pages 137, 141).

[110] Rodrigo Benenson. *Who is best at X?* http://rodrigob.github.io/are_we_there_yet/build/. 2017 (cited on page 163).

[111] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *J. Mach. Learn. Res.* 12 (July 2011), pages 2121–2159. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1953048.2021068 (cited on page 164).

[112] Matthew D. Zeiler. "ADADELTA: An Adaptive Learning Rate Method". In: *CoRR* abs/1212.5701 (2012). URL: http://arxiv.org/abs/1212.5701 (cited on page 164).

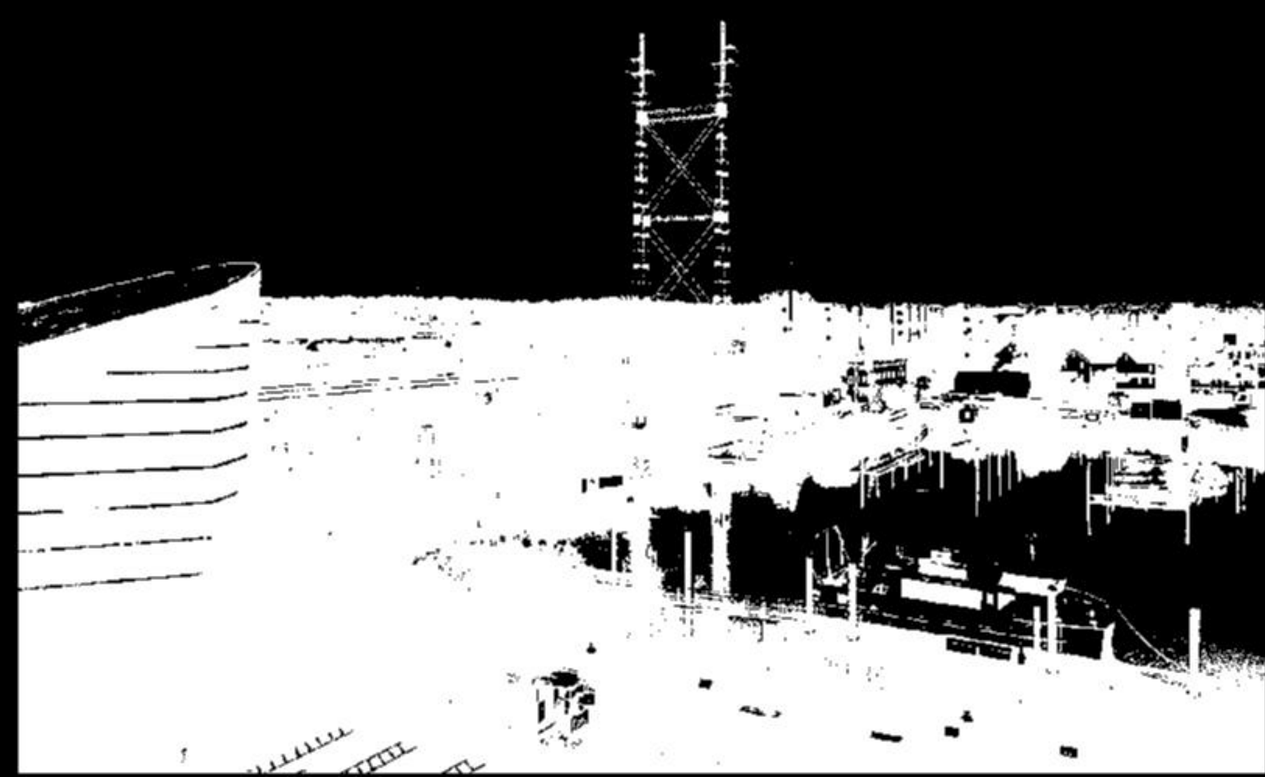

Figure 3.1: This image is 1,000 pixels wide and 750 pixels tall, for a total of $1,000 \times 750 = 750,000$ total pixels.

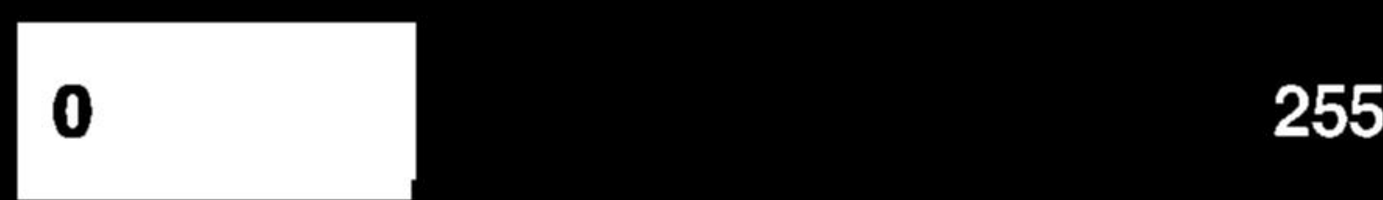

Figure 3.2: Image gradient demonstrating pixel values going from black (0) to white (255).

If you are interested in learning more about color spaces (and the fundamentals of computer vision and image processing), please see along with the .

Pixels in the RGB color space are no longer a scalar value like in a grayscale/single channel image – instead, the pixels are represented by a list of *three values*: one value for the Red component, one for Green, and another for Blue. To define a color in the RGB color model, all we need to do is define the amount of Red, Green, and Blue contained in a single pixel.

Each Red, Green, and Blue channel can have values defined in the range $[0, 255]$ for a total of 256 "shades", where 0 indicates no representation and 255 demonstrates full representation. Given that the pixel value only needs to be in the range $[0, 255]$, we normally use 8-bit unsigned integers to represent the intensity.

As we'll see once we build our first neural network, we'll often preprocess our image by performing mean subtraction or scaling, which will require us to convert the image to a floating point data type. Keep this point in mind as the data types used by libraries loading images from disk (such as OpenCV) will often need to be converted before we apply learning algorithms to the images directly.

Given our three Red, Green, and Blue values, we can combine them into an RGB tuple in the form `(red, green, blue)`. This tuple represents a given color in the RGB color space. The RGB color space is an example of an *additive* color space: the more of each color is added, the brighter the pixel becomes and closer to white. We can visualize the RGB color space in Figure 3.3 (*left*). As you can see, adding red and green leads to yellow. Adding red and blue yields pink. And

[113] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). URL: http://arxiv.org/abs/1412.6980 (cited on pages 164, 311).

[114] Greg Heinrich. *NVIDIA DIGITS: Weight Initialization*. https://github.com/NVIDIA/DIGITS/blob/master/examples/weight-init/README.md. 2015 (cited on pages 165, 167).

[115] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*. 2010 (cited on page 166).

[116] Andrew Jones. *An Explanation of Xavier Initialization*. http://andyljones.tumblr.com/post/110998971763/an-explanation-of-xavier-initialization. 2016 (cited on page 166).

[117] Kaiming He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *CoRR* abs/1502.01852 (2015). URL: http://arxiv.org/abs/1502.01852 (cited on page 167).

[118] Keras contributors. *Keras Initializers*. https://keras.io/initializers/#glorot_uniform. 2016 (cited on page 167).

[119] Richard Szeliski. *Computer Vision: Algorithms and Applications*. 1st. New York, NY, USA: Springer-Verlag New York, Inc., 2010. ISBN: 1848829345, 9781848829343 (cited on pages 171, 177).

[120] Victor Powell. *Image Kernels Explained Visually*. http://setosa.io/ev/image-kernels/. 2015 (cited on page 177).

[121] Andrej Karpathy. *Convolutional Networks*. http://cs231n.github.io/convolutional-networks/ (cited on pages 186, 187, 191).

[122] Jost Tobias Springenberg et al. "Striving for Simplicity: The All Convolutional Net". In: *CoRR* abs/1412.6806 (2014). URL: http://arxiv.org/abs/1412.6806 (cited on page 188).

[123] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167 (2015). URL: http://arxiv.org/abs/1502.03167 (cited on pages 189, 193).

[124] François Chollet. *BN Questions (old)*. https://github.com/fchollet/keras/issues/1802#issuecomment-187966878 (cited on page 190).

[125] Dmytro Mishkin. *CaffeNet-Benchmark – Batch Norm*. https://github.com/ducha-aiki/caffenet-benchmark/blob/master/batchnorm.md (cited on page 190).

[126] Reddit community contributors. *Batch Normalization before or after ReLU?* https://www.reddit.com/r/MachineLearning/comments/67gonq/d_batch_normalization_before_or_after_relu/ (cited on page 190).

[127] Forrest N. Iandola et al. "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size". In: *CoRR* abs/1602.07360 (2016). URL: http://arxiv.org/abs/1602.07360 (cited on pages 192, 280, 281).

[128] Pierre Sermanet et al. "OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks". In: *CoRR* abs/1312.6229 (2013). URL: http://arxiv.org/abs/1312.6229 (cited on page 229).

[129] Andrej Karpathy. *Neural Networks (Part I)*. http://cs231n.github.io/neural-networks-1/ (cited on page 253).

[130] Kaiming He et al. "Identity Mappings in Deep Residual Networks". In: *CoRR* abs/1603.05027 (2016). URL: http://arxiv.org/abs/1603.05027 (cited on page 279).

[131] Christian Szegedy et al. "Rethinking the Inception Architecture for Computer Vision". In: *CoRR* abs/1512.00567 (2015). URL: http://arxiv.org/abs/1512.00567 (cited on page 280).

[132] François Chollet. "Xception: Deep Learning with Depthwise Separable Convolutions". In: *CoRR* abs/1610.02357 (2016). URL: http://arxiv.org/abs/1610.02357 (cited on page 280).

[133] Wikipedia. *E-ZPass*. https://en.wikipedia.org/wiki/E-ZPass (cited on page 288).

[134] Paul Viola and Michael Jones. "Rapid object detection using a boosted cascade of simple features". In: 2001, pages 511–518 (cited on page 313).

adding all three red, green, and blue together, we create white.

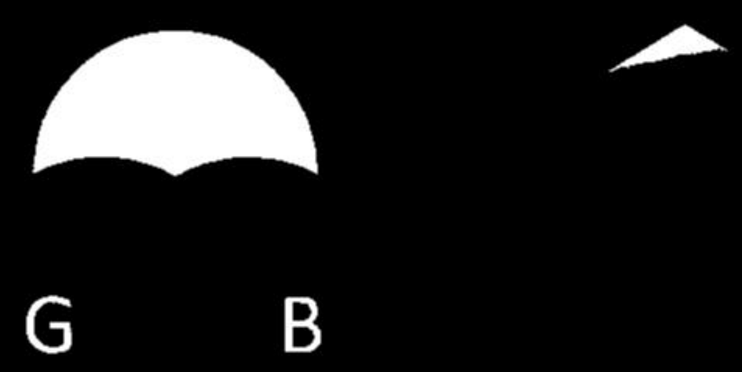

Figure 3.3: **Left:** The RGB color space is *additive*. The more red, green and blue you mix together, the closer you get to *white*. **Right:** The RGB cube.

To make this example more concrete, let's again consider the color "white" – we would fill each of the red, green, and blue buckets up completely, like this: `(255, 255, 255)`. Then, to create the color black, we would empty each of the buckets out `(0, 0, 0)`, as black is the absence of color. To create a pure red color, we would fill up the red bucket (and only the red bucket) completely: `(255, 0, 0)`.

The RGB color space is also commonly visualized as a cube (Figure 3.3, *right*) Since an RGB color is defined as a 3-valued tuple, which each value in the range $[0, 255]$ we can thus think of the cube containing $256 \times 256 \times 256 = 16,777,216$ possible colors, depending on how much Red, Green, and Blue are placed into each bucket.

As an example, let's consider how "much" red, green and blue we would need to create a single color (Figure 3.4, *top*). Here we set `R=252, G=198, B=188` to create a color tone similar to the skin of a caucasian (perhaps useful when building an application to detect the amount of skin/flesh in an image). As we can see, the Red component is heavily represented with the bucket almost filled. Green and Blue are represented almost equally. Combining these colors in an additive manner, we obtain a color tone similar to caucasian skin.

Figure 3.4: **Top:** An example of adding various Red, Green, and Blue color components together to create a "caucasian flesh tone", perhaps useful in a skin detection program. **Bottom:** Creating the specific shade of blue in the "PyImageSearch" logo by mixing various amounts of Red, Green, and Blue.

Let's try another example, this time setting `R=22, G=159, B=230` to obtain the shade of blue used in the PyImageSearch logo (Figure 3.4, *bottom*). Here Red is *heavily* under-represented with a

value of 22 – the bucket is so *empty* that the Red value actually appears to be "black". The Green bucket is a little over 50% full while the Blue bucket is over 90% filled, clearly the dominant color in the representation.

The primary drawbacks of the RGB color space include:

- Its additive nature makes it a bit unintuitive for humans to easily define shades of color without using a "color picker" tool.
- It doesn't mimic how humans perceive color.

Despite these drawbacks, nearly all images you'll work with will be represented (at least initially) in the RGB color space. For a full review of color models and color spaces, please refer to the [33].

Forming an Image From Channels

As we now know, an RGB image is represented by three values, one for each of the Red, Green, and Blue components, respectively. We can conceptualize an RGB image as consisting of *three independent matrices* of width `W` and height `H`, one for each of the RGB components, as shown in Figure 3.5. We can combine these three matrices to obtain a multi-dimensional array with shape $W \times H \times D$ where `D` is the **depth** or **number of channels** (for the RGB color space, `D=3`):

Figure 3.5: Representing an image in the RGB color space where each channel is an independent matrix, that when combined, forms the final image.

Keep in mind that the **depth** of an image is *very different* than the **depth** of a neural network – this point will become clear when we start training our own Convolutional Neural Networks. However, for the time being simply understand that the vast majority of the images you'll be working with are:

- Represented in the RGB color space by three channels, each channel in the range $[0, 255]$. A given pixel in an RGB image is a list of three integers: one value for Red, second for Green, and a final value for Blue.
- Programmatically defined as a 3D NumPy multidimensional arrays with a width, height, and depth.

The Image Coordinate System

As mentioned in Figure 3.1 earlier in this chapter, an image is represented as a grid of pixels. To make this point more clear, imagine our grid as a piece of graph paper. Using this graph paper, the origin point $(0, 0)$ corresponds to the **upper-left** corner of the image. As we move down and to the right, both the x and y values increase.

Figure 3.6 provides a visual representation of this "graph paper" representation. Here we have the letter "I" on a piece of our graph paper. We see that this is an 8×8 grid with a total of 64 pixels.

It's important to note that we are counting from *zero* rather than *one*. The Python language is *zero indexed*, meaning that we always start counting from zero. Keep this in mind as you'll avoid a lot of confusion later on (especially if you're coming from a MATLAB environment).

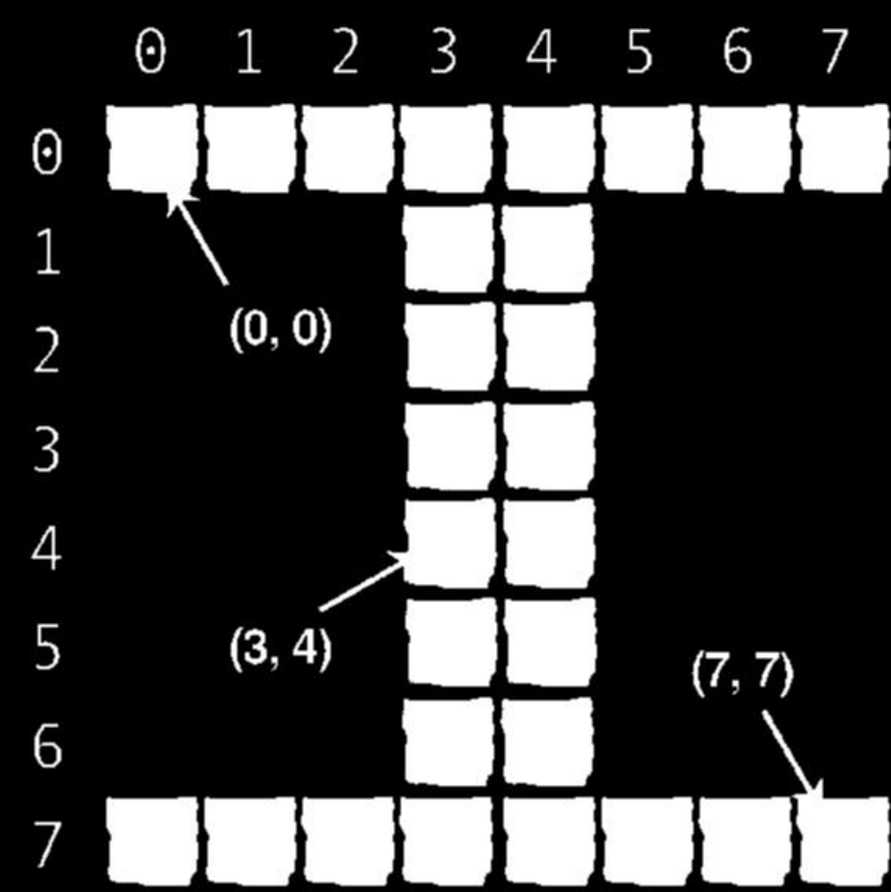

Figure 3.6: The letter "I" placed on a piece of graph paper. Pixels are accessed by their (x, y)-coordinates, where we go x columns to the right and y rows down, keeping in mind that Python is zero-indexed.

As an example of zero-indexing, consider the pixel 4 columns to the right and 5 rows down is indexed by the point $(3, 4)$, again keeping in mind that we are counting from *zero* rather than *one*.

Images as NumPy Arrays

Figure 3.7: Loading an image named `example.png` from disk and displaying it to our screen with OpenCV.

Image processing libraries such as OpenCV and scikit-image represent RGB images as multi-dimensional NumPy arrays with shape `(height, width, depth)`. Readers who are using image processing libraries for the first time are often confused by this representation – why does the *height* come before the *width* when we normally think of an image in terms of width *first* then height?

The answer is due to matrix notation.

When defining the dimensions of matrix, we always write it as `rows x columns`. The number of `rows` in an image is its height whereas the number of `columns` is the image's width. The depth will still remain the depth.

Therefore, while it may be slightly confusing to see the `.shape` of a NumPy array represented as `(height, width, depth)`, this representation actually makes intuitive sense when considering how a matrix is constructed and annotated.

For example, let's take a look at the OpenCV library and the `cv2.imread` function used to load an image from disk and display its dimensions:

```
        cv2
image = cv2.imread("example.png")
     (image.shape)
cv2.imshow("Image", image)
cv2.waitKey( )
```

Here we load an image named `example.png` from disk and display it to our screen, as the screenshot from Figure 3.7 demonstrates. My terminal output follows:

```
python load_display.py
```

This image has a width of 300 pixels (the number of columns), a height of 248 pixels (the number of rows), and a depth of 3 (the number of channels). To access an individual pixel value from our `image` we use simple NumPy array indexing:

```
(b, g, r) = image[  ,    ]
(b, g, r) = image[  ,   ]
(b, g, r) = image[  ,   ]
```

Again, notice how the `y` value is passed in *before* the `x` value – this syntax may feel uncomfortable at first, but it is consistent with how we access values in a matrix: first we specify the row number then the column number. From there, we are given a tuple representing the Red, Green, and Blue components of the image.

RGB and BGR Ordering

It's important to note that OpenCV stores RGB channels in *reverse order*. While we normally think in terms of Red, Green, and Blue, OpenCV actually stores the pixel values in Blue, Green, Red order.

Why does OpenCV do this? The answer is simply historical reasons. Early developers of the OpenCV library chose the BGR color format because the BGR ordering was popular among camera manufacturers and other software developers at the time [41].

Simply put – this BGR ordering was made for historical reasons and a choice that we now have to live with. It's a small caveat, but an important one to keep in mind when working with OpenCV.

Scaling and Aspect Ratios

Scaling, or simply *resizing*, is the process of increasing or decreasing the size of an image in terms of width and height. When resizing an image, it's important to keep in mind the *aspect ratio*, which

Figure 3.8: **Left:** Original image. **Top and Bottom:** Resulting distorted images after resizing without preserving the aspect ratio (i.e., the ratio of the width to the height of the image).

is the ratio of the width to the height of the image. Ignoring the aspect ratio can lead to images that look compressed and distorted, as in Figure 3.8.

On the *left*, we have the original image. And on the *top and bottom*, we have two images that have been distorted by not preserving the aspect ratio. The end result is that these images are distorted, crunched, and squished. To prevent this behavior, we simply scale the width and height of an image by equal amounts when resizing an image.

From a strictly *aesthetic* point of view, you almost always want to ensure the aspect ratio of the image is maintained when resizing an image – **but this guideline isn't always the case for deep learning**. Most neural networks and Convolutional Neural Networks applied to the task of image classification assume a *fixed size input*, meaning that the dimensions of *all images* you pass through the network *must be the same*. Common choices for width and height image sizes inputted to Convolutional Neural Networks include 32×32, 64×64, 224×224, 227×227, 256×256, and 299×299.

Let's assume we are designing a network that will need to classify 224×224 images; however, our dataset consists of images that are 312×234, 800×600, and 770×300, among other image sizes – how are we supposed to preprocess these images? Do we simply ignore the aspect ratio and deal with the distortion (Figure 3.9, *bottom left*)? Or do we devise another scheme to resize the image, such as resizing the image along its shortest dimension and then taking the center crop (Figure 3.9, *bottom right*)?

As we can see in in the *bottom left*, the aspect ratio of our image has been ignored, resulting in an image that looks distorted and "crunched". Then, in the *bottom right*, we see that the aspect ratio of the image has been maintained, but at the expense of cropping out part of the image. This could be especially detrimental to our image classification system if we accidentally crop part or all of the object we wish to identify.

Which method is best? **In short, *it depends*.** For some datasets you can simply ignore the aspect ratio and squish, distort, and compress your images prior to feeding them through your network. On other datasets, it's advantageous to preprocess them further by resizing along the shortest dimension and then cropping the center.

We'll be reviewing both of these methods (and how to implement them) in more detail later in this book, but it's important to introduce this topic now as we study the fundamentals of images

Figure 3.9: **Top:** Our original input image. **Bottom left:** Resizing an image to 224×224 pixels by ignoring the aspect ratio. **Bottom right:** Resizing an image 224×224 pixels by first resizing along the shortest dimension and then taking the center crop.

and how they are represented.

Summary

This chapter reviewed the fundamental building blocks of an image – the pixel. We learned that grayscale/single channel images are represented by a single scalar, the intensity/brightness of the pixel. The most common color space is the RGB color space where each pixel in an image is represented by a 3-tuple: one for each of the Red, Green, and Blue components, respectively.

We also learned that computer vision and image processing libraries in the Python programming language leverage the powerful NumPy numerical processing library and thus represent images as multi-dimensional NumPy arrays. These arrays have the shape `(height, width, depth)`.

The height is specified first because the height is the number of rows in the matrix. The width comes next, as it is the number of columns in the matrix. Finally, the depth controls the number of channels in the image. In the RGB color space, the depth is fixed at `depth=3`.

Finally, we wrapped up this chapter by reviewing the *aspect ratio* of an image and the role it will play when we resize images as inputs to our neural networks and Convolutional Neural Networks. For a more detailed review of color spaces, the image coordinate system, resizing/aspect ratios, and other basics of the OpenCV library, please refer to [8] and the [33].

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

4. Image Classification Basics

"A picture is worth a thousand words" – English idiom

We've heard this adage countless times in our lives. It simply means that a complex idea can be conveyed in a single image. Whether examining the line chart of our stock portfolio investments, looking at the spread of an upcoming football game, or simply taking in the art and brush strokes of a painting master, we are constantly ingesting visual content, interpreting the meaning, and storing the knowledge for later use.

However, for computers, interpreting the contents of an image is less trivial – all our computer sees is a big matrix of numbers. It has *no idea* regarding the thoughts, knowledge, or meaning the image is trying to convey.

In order to understand the contents of an image, we must apply ***image classification***, which is the task of using computer vision and machine learning algorithms to extract meaning from an image. This action could be as simple as assigning a label to what the image contains, or as advanced as interpreting the contents of an image and returning a human-readable sentence.

Image classification is a very large field of study, encompassing a wide variety of techniques – and with the popularity of deep learning, it is continuing to grow.

Now is the time to ride the deep learning and image classification wave – those who successfully do so will be handsomely rewarded.

Image classification and image understanding are currently (and will continue to be) the most popular sub-field of computer vision for the next ten years. In the future, we'll see companies like Google, Microsoft, Baidu, and others quickly acquire successful image understanding startup companies. We'll see more and more consumer applications on our smartphones that can understand and interpret the contents of an image. Even wars will likely be fought using unmanned aircrafts that are *automatically* guided using computer vision algorithms.

Inside this chapter, I'll provide a high-level overview of what image classification is, along with the many challenges an image classification algorithm has to overcome. We'll also review the three different types of learning associated with image classification and machine learning.

Finally, we'll wrap up this chapter by discussing the four steps of training a deep learning network for image classification and how this four step pipeline compares to the *traditional*,

hand-engineered feature extraction pipeline.

What Is Image Classification?

Image classification, at its very core, is the task of *assigning a label to an image* from a *predefined set of categories*.

Practically, this means that our task is to analyze an input image and return a label that categorizes the image. The label is always from a predefined set of possible categories.

For example, let's assume that our set of possible categories includes:

`categories = {cat, dog, panda}`

Then we present the following image (Figure 4.1) to our classification system:

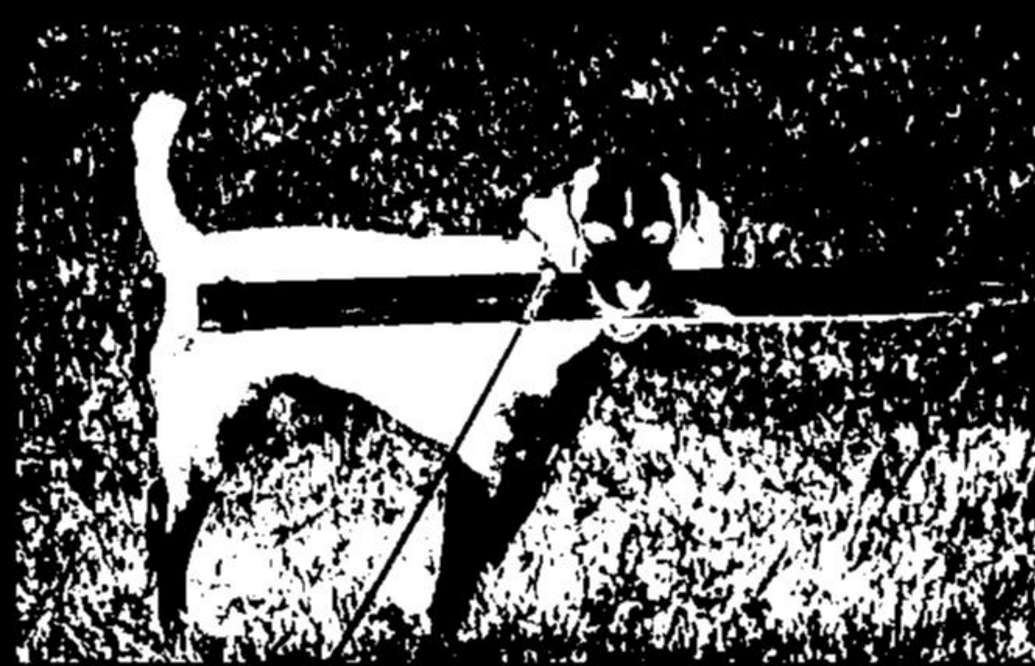

Figure 4.1: The goal of an image classification system is to take an input image and assign a label based on a pre-defined set of categories.

Our goal here is to take this input image and assign a label to it from our **categories** set – in this case, **dog**.

Our classification system could also assign multiple labels to the image via probabilities, such as **dog: 95%**; **cat: 4%**; **panda: 1%**.

More formally, given our input image of $W \times H$ pixels with three channels, Red, Green, and Blue, respectively, our goal is to take the $W \times H \times 3 = N$ pixel image and figure out how to correctly classify the contents of the image.

A Note on Terminology

When performing machine learning and deep learning, we have a ***dataset*** we are trying to extract knowledge from. Each example/item in the dataset (whether it be image data, text data, audio data, etc.) is a ***data point***. A dataset is therefore a collection of data points (Figure 4.2).

Our goal is to apply a machine learning and deep learning algorithms to discover underlying patterns in the dataset, enabling us to correctly classify data points that our algorithm has not encountered yet. Take the time now to familiarize yourself with this terminology:

1. In the context of image classification, our *dataset* is a *collection of images*.
2. Each *image* is, therefore, a *data point*.

I'll be using the term *image* and *data point* interchangeably throughout the rest of this book, so keep this in mind now.

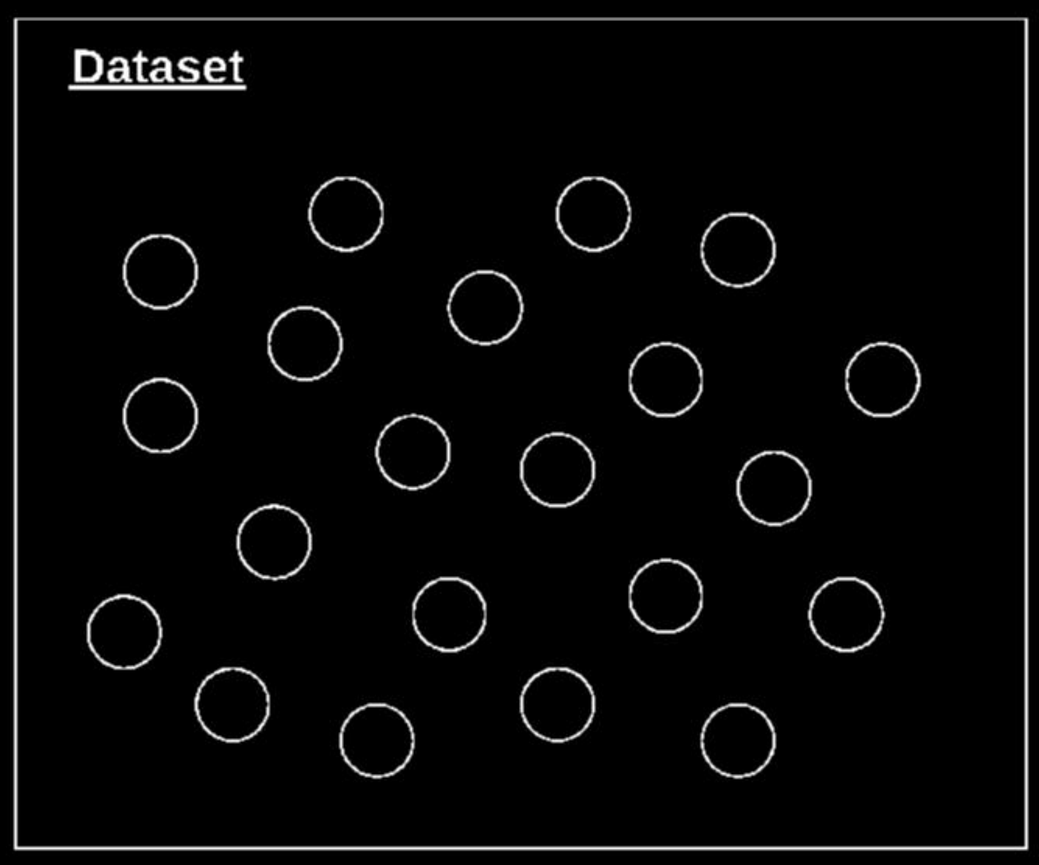

Figure 4.2: A dataset (outer rectangle) is a *collection* of data points (circles).

The Semantic Gap

Take a look at the two photos (*top*) in Figure 4.3. It should be fairly trivial for us to tell the difference between the two photos – there is clearly a *cat* on the left and a *dog* on the right. But all a computer sees is two big matrices of pixels (*bottom*).

Given that all a computer sees is a big matrix of pixels, we arrive at the problem of the *semantic gap*. The semantic gap is the difference between how a human *perceives* the contents of an image versus how an image can be *represented* in a way a computer can understand the process.

Again, a quick visual examination of the two photos above can reveal the difference between the two species of an animal. But in reality, the computer has *no idea* there are animals in the image to begin with. To make this point clear, take a look at Figure 4.4, containing a photo of a tranquil beach.

We might describe the image as follows:

- **Spatial:** The sky is at the top of the image and the sand/ocean are at the bottom.
- **Color:** The sky is dark blue, the ocean water is a lighter blue than the sky, while the sand is tan.
- **Texture:** The sky has a relatively uniform pattern, while the sand is very coarse.

How do we go about encoding all this information in a way that a computer can understand it? The answer is to apply *feature extraction* to quantify the contents of an image. Feature extraction is the process of taking an input image, applying an algorithm, and obtaining a feature vector (i.e., a list of numbers) that quantifies our image.

To accomplish this process, we may consider applying hand-engineered features such as HOG, LBPs, or other "traditional" approaches to image quantifying. Another method, and the one taken by this book, is to apply deep learning to automatically *learn a set of features* that can be used to quantify and ultimately *label* the contents of the image itself.

However, it's not that simple. . . because once we start examining images in the real world, we are faced with many, *many* challenges.

Figure 4.3: **Top:** Our brains can clearly see the difference between an image that contains a *cat* and an image that contains a *dog*. **Bottom:** However, all a computer "sees" is a big matrix of numbers. The difference between how we perceive an image and how the image is represented (a matrix of numbers) is called the *semantic gap*.

Challenges

If the semantic gap were not enough of a problem, we also have to handle **factors of variation** [10] in how an image or object appears. Figure 4.5 displays a visualization of a number of these factors of variation.

To start, we have ***viewpoint variation***, where an object can be oriented/rotated in multiple dimensions with respect to how the object is photographed and captured. No matter the angle in which we capture this Raspberry Pi, it's still a Raspberry Pi.

We also have to account for ***scale variation*** as well. Have you ever ordered a tall, grande, or venti cup of coffee from Starbucks? Technically they are all the same thing – a cup of coffee. But they are all different *sizes* of a cup of coffee. Furthermore, that same venti coffee will look dramatically different when it is photographed up close versus when it is captured from farther away. Our image classification methods must be tolerable to these types of scale variations.

One of the hardest variations to account for is ***deformation***. For those of you familiar with the television series *Gumby*, we can see the main character in the image above. As the name of the TV show suggests, this character is elastic, stretchable, and capable of contorting his body in many different poses. We can look at these images of *Gumby* as a type of *object deformation* – all images contain the *Gumby* character; however, they are all dramatically different from each other.

Our image classification should also be able to handle ***occlusions***, where large parts of the

Figure 4.4: When describing the contents of this image we may focus on words that the convey the *spatial layout*, *color*, and *texture* – the same is true for computer vision algorithms.

object we want to classify are hidden from view in the image (Figure 4.5). On the *left* we have to have a picture of a dog. And on the *right* we have a photo of the same dog, but notice how the dog is resting underneath the covers, *occluded* from our view. The dog is still clearly in both images – she's just more visible in one image than the other. Image classification algorithms should still be able to detect and label the presence of the dog in both images.

Just as challenging as the *deformations* and *occlusions* mentioned above, we also need to handle the changes in ***illumination***. Take a look at the coffee cup captured in standard and low lighting (Figure 4.5). The image on the *left* was photographed with standard overhead lighting while the image on the *right* was captured with very little lighting. We are still examining the same cup – but based on the lighting conditions, the cup looks dramatically different (nice how the vertical cardboard seam of the cup is clearly visible in the low lighting conditions, but not the standard lighting).

Continuing on, we must also account for ***background clutter***. Ever play a game of *Where's Waldo?* (Or *Where's Wally?* for our international readers.) If so, then you know the goal of the game is to find our favorite red-and-white, striped shirt friend. However, these puzzles are more than just an entertaining children's game – they are also the perfect representation of *background clutter*. These images are incredibly "noisy" and have a lot going on in them. We are only interested in *one* particular object in the image; however, due to all the "noise", it's not easy to pick out Waldo/Wally. If it's not easy for us to do, imagine how hard it is for a computer with no semantic understanding of the image!

Finally, we have ***intra-class variation***. The canonical example of intra-class variation in computer vision is displaying the diversification of chairs. From comfy chairs that we use to curl up and read a book, to chairs that line our kitchen table for family gatherings, to ultra-modern art deco chairs found in prestigious homes, a chair is still a chair – and our image classification algorithms must be able to categorize all these variations correctly.

Are you starting to feel a bit overwhelmed with the complexity of building an image classifier? Unfortunately, it only gets worse – it's not enough for our image classification system to be robust to these variations *independently*, but our system must also handle *multiple variations combined together!*

So how do we account for such an incredible number of variations in objects/images? In general, we try to frame the problem as best we can. We make assumptions regarding the contents of our images and to which variations we want to be tolerant. We also consider the scope of our

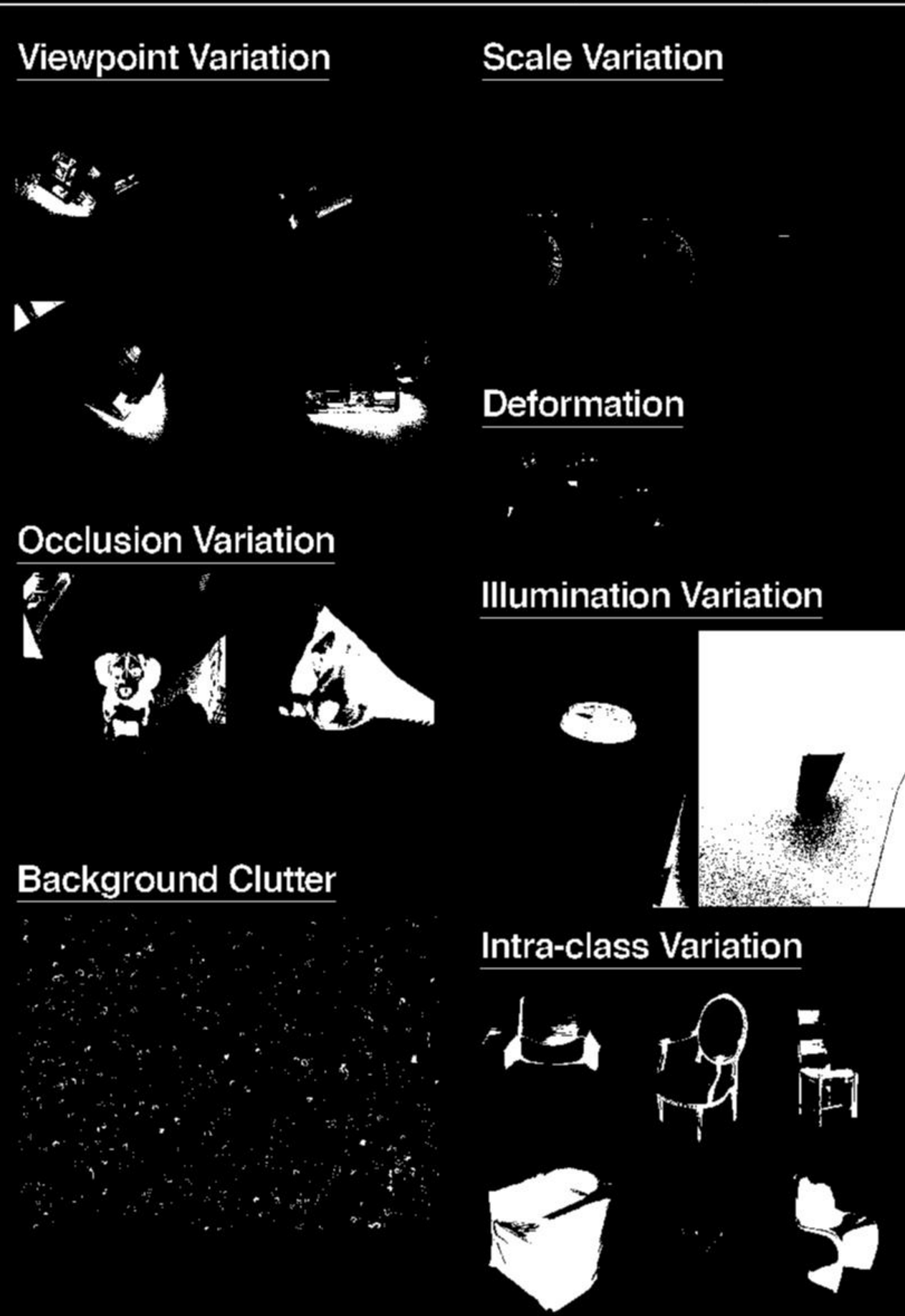

Figure 4.5: When developing an image classification system, we need to be cognizant of how an object can appear at varying viewpoints, lighting conditions, occlusions, scale, etc.

project – what is the end goal? And what are we trying to build?

Successful computer vision, image classification, and deep learning systems deployed to the real-world make *careful assumptions and considerations* before a single line of code is ever written.

If you take too broad of an approach, such as *"I want to classify and detect every single object in my kitchen"*, (where there could be hundreds of possible objects) then your classification system is unlikely to perform well unless you have years of experience building image classifiers – and even then, there is no guarantee to the success of the project.

But if you **frame your problem** and make it narrow in scope, such as *"I want to recognize **just** stoves and refrigerators"*, then your system is **much more likely** to be accurate and functioning, *especially* if this is your first time working with image classification and deep learning.

The key takeaway here is to ***always consider the scope of your image classifier***. While deep learning and Convolutional Neural Networks have demonstrated significant robustness and classification power under a variety of challenges, you *still* should keep the scope of your project as

tight and well-defined as possible.

Keep in mind that ImageNet [42], the *de facto* standard benchmark dataset for image classification algorithms, consists of 1,000 objects that we encounter in our everyday lives – and this dataset is *still* actively used by researchers trying to push the state-of-the-art for deep learning forward.

Deep learning is *not* magic. Instead, deep learning is like a scroll saw in your garage – powerful and useful when wielded correctly, but hazardous if used without proper consideration. Throughout the rest of this book, I will guide you on your deep learning journey and help point out when you should reach for these power tools and when you should instead refer to a simpler approach (or mention if a problem isn't reasonable for image classification to solve).

Types of Learning

There are three types of learning that you are likely to encounter in your machine learning and deep learning career: supervised learning, unsupervised learning, and semi-supervised learning. This book focuses mostly on supervised learning in the context of deep learning. Nonetheless, descriptions of all three types of learning are presented below.

Supervised Learning

Imagine this: you've just graduated from college with your Bachelors of Science in Computer Science. You're young. Broke. And looking for a job in the field – perhaps you even feel lost in your job search.

But before you know it, a Google recruiter finds you on LinkedIn and offers you a position working on their Gmail software. Are you going to take it? Most likely.

A few weeks later, you pull up to Google's spectacular campus in Mountain View, California, overwhelmed by the breathtaking landscape, the fleet of Teslas in the parking lot, and the almost never-ending rows of gourmet food in the cafeteria.

You finally sit down at your desk in a wide-open workspace among hundreds of other employees... *and then you find out your role in the company*. You've been hired to create a piece of software to *automatically classify* email as *spam* or *not-spam*.

How are going to accomplish this goal? Would a rule-based approach work? Could you write a series of `if/else` statements that look for certain words and then determine if an email is spam based on these rules? That might work... to a degree. But this approach would also be easily defeated and near impossible to maintain.

Instead, what you *really need* is machine learning. You need a *training set* consisting of the emails themselves along with their *labels*, in this case *spam* or *not-spam*. Given this data, you can analyze the text (i.e., the distributions of words) in the email and utilize the spam/not-spam labels to teach a machine learning classifier what words occur in a spam email and which do not – all without having to manually create a long and complicated series of `if/else` statements.

This example of creating a spam filter system is an example of **supervised learning**. Supervised learning is arguably the most well known and studied type of machine learning. Given our training data, a model (or "classier") is created through a training process where predictions are made on the input data and then corrected when the predictions are wrong. This training process continues until the model achieves some desired stopping criterion, such as a low error rate or a maximum number of training iterations.

Common supervised learning algorithms include Logistic Regression, Support Vector Machines (SVMs) [43, 44], Random Forests [45], and Artificial Neural Networks.

In the context of **image classification**, we assume our image dataset consists of the images themselves along with their corresponding *class label* that we can use to teach our machine learning

Label	R_μ	G_μ	B_μ	R_σ	G_σ	B_σ
Cat	57.61	41.36	123.44	158.33	149.86	93.33
Cat	120.23	121.59	181.43	145.58	69.13	116.91
Cat	124.15	193.35	65.77	23.63	193.74	162.70
Dog	100.28	163.82	104.81	19.62	117.07	21.11
Dog	177.43	22.31	149.49	197.41	18.99	187.78
Dog	149.73	87.17	187.97	50.27	87.15	36.65

Table 4.1: A table of data containing both the class labels (either *dog* or *cat*) and feature vectors for each data point (the mean and standard deviation of each Red, Green, and Blue color channel, respectively). This is an example of a ***supervised classification*** task.

classifier what each category "looks like". If our classifier makes an incorrect prediction, we can then apply methods to correct its mistake.

The differences between supervised, unsupervised, and semi-supervised learning can best be understood by looking at the example in Table 4.1. The first column of our table is the label associated with a particular image. The remaining six columns correspond to our feature vector for each data point – here, we have chosen to quantify our image contents by computing the mean and standard deviation for each RGB color channel, respectively.

Our supervised learning algorithm will make predictions on each of these feature vectors, and if it makes an incorrect prediction, we'll attempt to correct it by telling it what the correct label actually is. This process will then continue until the desired stopping criterion has been met, such as accuracy, number of iterations of the learning process, or simply an arbitrary amount of wall time.

> To explain the differences between supervised, unsupervised, and semi-supervised learning, I have chosen to use a feature-based approach (i.e., the mean and standard deviation of the RGB color channels) to quantify the content of an image. When we start working with Convolutional Neural Networks, we'll actually **skip** the feature extraction step and use the raw pixel intensities themselves. Since images can be large $M \times N$ matrices (and therefore cannot fit nicely into this spreadsheet/table example), I have used the feature-extraction process to help visualize the differences between types of learning.

Unsupervised Learning

In contrast to supervised learning, **unsupervised learning** (sometimes called **self-taught learning**) has no labels associated with the input data and thus we cannot correct our model if it makes an incorrect prediction.

Going back to the spreadsheet example, converting a supervised learning problem to an unsupervised learning one is as simple as removing the "label" column (Table 4.2).

Unsupervised learning is sometimes considered the "holy grail" of machine learning and image classification. When we consider the number of images on Flickr or the number of videos on YouTube, we quickly realize there is a *vast amount* of unlabeled data available on the internet. If we could get our algorithm to learn patterns from *unlabeled data*, then we wouldn't have to spend large amounts of time (and money) arduously labeling images for supervised tasks.

Most unsupervised learning algorithms are most successful when we can learn the underlying structure of a dataset and then, in turn, apply our learned features to a *supervised* learning problem where there is too little labeled data to be of use.

Classic machine learning algorithms for unsupervised learning include Principle Component Analysis (PCA) and k-means clustering. Specific to neural networks, we see Autoencoders, Self-

R_μ	G_μ	B_μ	R_σ	G_σ	B_σ
57.61	41.36	123.44	158.33	149.86	93.33
120.23	121.59	181.43	145.58	69.13	116.91
124.15	193.35	65.77	23.63	193.74	162.70
100.28	163.82	104.81	19.62	117.07	21.11
177.43	22.31	149.49	197.41	18.99	187.78
149.73	87.17	187.97	50.27	87.15	36.65

Table 4.2: Unsupervised learning algorithms attempt to learn underlying patterns in a dataset *without* class labels. In this example we have removed the class label column, thus turning this task into an ***unsupervised learning*** problem.

Label	R_μ	G_μ	B_μ	R_σ	G_σ	B_σ
Cat	57.61	41.36	123.44	158.33	149.86	93.33
?	120.23	121.59	181.43	145.58	69.13	116.91
?	124.15	193.35	65.77	23.63	193.74	162.70
Dog	100.28	163.82	104.81	19.62	117.07	21.11
?	177.43	22.31	149.49	197.41	18.99	187.78
Dog	149.73	87.17	187.97	50.27	87.15	36.65

Table 4.3: When performing ***semi-supervised learning*** we only have the labels for a subset of the images/feature vectors and must try to label the other data points to utilize them as extra training data.

Organizing Maps (SOMs), and Adaptive Resonance Theory applied to unsupervised learning. Unsupervised learning is an extremely active area of research and one that has yet to be solved. We do not focus on unsupervised learning in this book.

Semi-supervised Learning

So, what happens if we only have *some* of the labels associated with our data and *no labels* for the other? Is there a way we can apply some hybrid of supervised and unsupervised learning and still be able to classify each of the data points? It turns out the answer is *yes* – we just need to apply semi-supervised learning.

Going back to our spreadsheet example, let's say we only have labels for a small fraction of our input data (Table 4.3). Our semi-supervised learning algorithm would take the known pieces of data, analyze them, and try to label each of the unlabeled data points for use as *additional* training data. This process can repeat for many iterations as the semi-supervised algorithm learns the "structure" of the data to make more accurate predictions and generate more reliable training data.

Semi-supervised learning is especially useful in computer vision where it is often time-consuming, tedious, and expensive (at least in terms of man-hours) to label each and every single image in our training set. In cases where we simply do not have the time or resources to label each individual image, we can label only a tiny fraction of our data and utilize semi-supervised learning to label and classify the rest of the images.

Semi-supervised learning algorithms often trade smaller labeled input datasets for some tolerable reduction in classification accuracy. Normally, the more accurately-labeled training a supervised learning algorithm has, the more accurate predictions it can make (this is *especially* true for deep learning algorithms).

As the amount of training data decreases, accuracy inevitably suffers. Semi-supervised learning

takes this relationship between accuracy and amount of data into account and attempts to keep classification accuracy within tolerable limits while dramatically reducing the amount of training data required to build a model – the end result is an accurate classifier (but normally not as accurate as a supervised classifier) with less effort and training data. Popular choices for semi-supervised learning include label spreading [46], label propagation [47], ladder networks [48], and co-learning/co-training [49].

Again, we'll primarily be focusing on supervised learning inside this book, as both unsupervised and semi-supervised learning in the context of deep learning for computer vision are still very active research topics without clear guidelines on which methods to use.

The Deep Learning Classification Pipeline

Based on our previous two sections on image classification and types of learning algorithms, you might be starting to feel a bit steamrolled with new terms, considerations, and what looks to be an insurmountable amount of variation in building an image classifier, but the truth is that building an image classifier is fairly straightforward, *once you understand the process*.

In this section we'll review an important shift in mindset you need to take on when working with machine learning. From there I'll review the four steps of building a deep learning-based image classifier as well as compare and contrast traditional feature-based machine learning versus end-to-end deep learning.

A Shift in Mindset

Before we get into anything complicated, let's start off with something that we're all (most likely) familiar with: *the Fibonacci sequence*.

The Fibonacci sequence is a series of numbers where the next number of the sequence is found by summing the two integers before it. For example, given the sequence `0, 1, 1`, the next number is found by adding `1 + 1 = 2`. Similarly, given `0, 1, 1, 2`, the next integer in the sequence is `1 + 2 = 3`.

Following that pattern, the first handful of numbers in the sequence are as follows:

`0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...`

Of course, we can also define this pattern in an (extremely unoptimized) Python function using recursion:

```
>>>         (n):
...         n ==   :
...
...           n ==   :
...
...          :
...                     fib(n- ) + fib(n- )
...
>>>
```

Using this code, we can compute the *n*-th number in the sequence by supplying a value of `n` to the `fib` function. For example, let's compute the 7th number in the Fibonacci sequence:

```
>>> fib( )

```

And the 13th number:

```
>>> fib(  )

```

And finally the 35th number:

```
>>> fib(  )

```

As you can see, the Fibonacci sequence is straightforward and is an example of a family of functions that:

1. Accepts an input, returns an output.
2. The process is well defined.
3. The output is easily verifiable for correctness.
4. Lends itself well to code coverage and test suites.

In general, you've probably written thousands upon thousands of procedural functions like these in your life. Whether you're computing a Fibonacci sequence, pulling data from a database, or calculating the mean and standard deviation from a list of numbers, these functions are all well defined and easily verifiable for correctness.

Unfortunately, this is not the case for deep learning and image classification!

Recall from Section 4.1.2 where we looked at the pictures of a cat and a dog, replicated in Figure 4.6 for convenience. Now, imagine trying to write a procedural function that can not only tell the difference between *these two photos*, but *any* photo of a cat and a dog. How would you go about accomplishing this task? Would you check individual pixel values at various (x, y)-coordinates? Write hundreds of `if/else` statements? And how would you maintain and verify the correctness of such as massive rule-based system? The short answer is: **you don't**.

Figure 4.6: How might you go about writing a piece of software to recognize the difference between dogs and cats in images? Would you inspect individual pixel values? Take a rule-based approach? Try to write (and maintain) hundreds of *if/else* statements?

Unlike coding up an algorithm to compute the Fibonacci sequence or sort a list of numbers, it's not intuitive or obvious how to create an algorithm to tell the difference between pictures of cats and dogs. Therefore, instead of trying to construct a rule-based system to describe what each category "looks like", we can instead take a *data driven approach* by supplying *examples* of what each category looks like and then *teach* our algorithm to recognize the difference between the categories using these examples.

We call these examples our *training dataset* of labeled images, where each data point in our training dataset consists of:

1. An image

2. The label/category (i.e., dog, cat, panda, etc.) of the image

Again, it's important that each of these images have labels associated with them because our supervised learning algorithm will need to see these labels to "teach itself" how to recognize each category. Keeping this in mind, let's go ahead and work through the four steps to constructing a deep learning model.

Step #1: Gather Your Dataset

The first component of building a deep learning network is to gather our initial dataset. We need the *images themselves* as well as the *labels* associated with each image. These labels should come from a finite set of categories, such as: `categories = dog, cat, panda`.

Furthermore, the number of images for each category should be approximately uniform (i.e., the same number of examples per category). If we have twice the number of cat images than dog images, and five times the number of panda images than cat images, then our classifier will become naturally biased to overfitting into these heavily-represented categories.

Class imbalance is a common problem in machine learning and there exist a number of ways to overcome it. We'll discuss some of these methods later in this book, but keep in mind the best method to avoid learning problems due to class imbalance is to simply avoid class imbalance entirely.

Step #2: Split Your Dataset

Now that we have our initial dataset, we need to split it into two parts:

1. A *training set*
2. A *testing set*

A *training set* is used by our classifier to "learn" what each category looks like by making predictions on the input data and then correct itself when predictions are wrong. After the classifier has been trained, we can evaluate the performing on a *testing set*.

It's extremely important that the training set and testing set are *independent of each other* and *do not overlap!* If you use your testing set as part of your training data, then your classifier has an unfair advantage since it has already seen the testing examples before and "learned" from them. Instead, you must keep this testing set entirely separate from your training process and use it ***only to evaluate your network***.

Common split sizes for training and testing sets include $66.\overline{6}\%33.\overline{3}\%$, $75\%/25\%$, and $90\%/10\%$, respectively. (Figure 4.7):

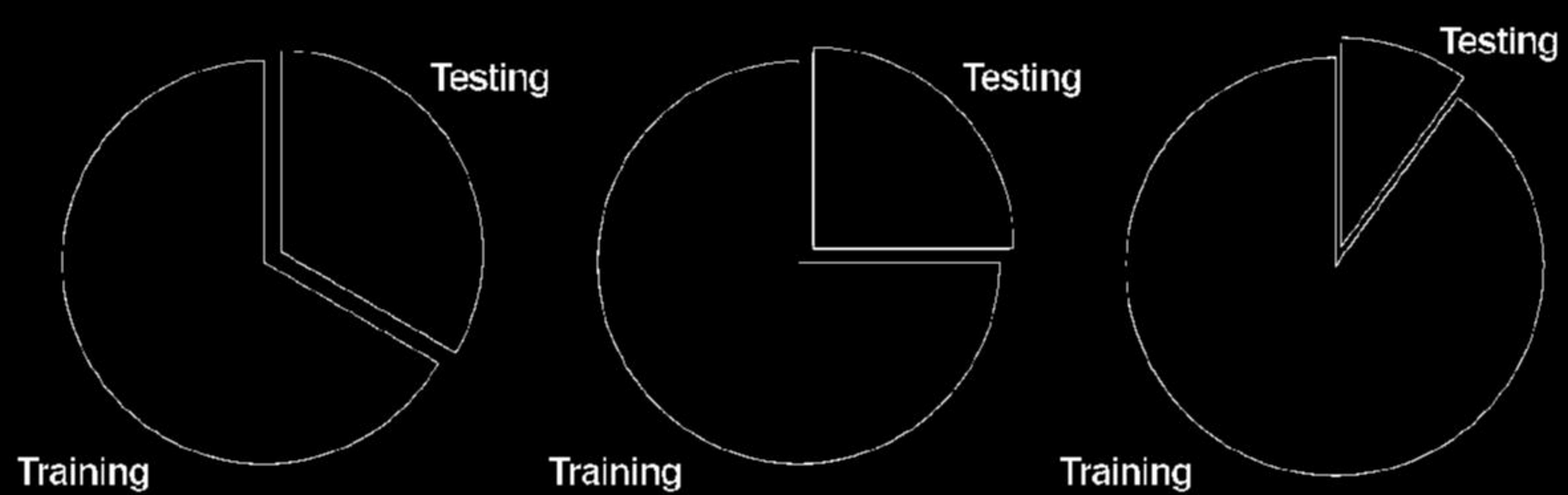

Figure 4.7: Examples of common training and testing data splits.

These data splits make sense, **but what if you have parameters to tune?** Neural networks have a number of knobs and levers (ex., learning rate, decay, regularization, etc.) that need to be tuned

and dialed to obtain optimal performance. We'll call these types of parameters *hyperparameters*, and it's *critical* that they get set properly.

In practice, we need to test a bunch of these hyperparameters and identify the set of parameters that works the best. You might be tempted to use your testing data to tweak these values, ***but again, this is a major no-no!*** The test set is *only* used in evaluating the performance of your network.

Instead, you should create a *third* data split called the **validation set**. This set of the data (normally) comes from the training data and is used as "fake test data" so we can tune our hyperparameters. Only after have we determined the hyperparameter values using the validation set do we move on to collecting final accuracy results in the testing data.

We normally allocate roughly 10-20% of the training data for validation. If splitting your data into chunks sounds complicated, it's actually not. As we'll see in our next chapter, it's quite simple and can be accomplished with only a **single line of code** thanks to the scikit-learn library.

Step #3: Train Your Network

Given our training set of images, we can now train our network. The goal here is for our network to learn how to recognize each of the categories in our labeled data. When the model makes a mistake, it learns from this mistake and improves itself.

So, how does the actual "learning" work? In general, we apply a form of gradient descent, as discussed in Chapter 9. The remainder of this book is dedicated to demonstrating how to train neural networks from scratch, so we'll defer a detailed discussion of the training process until then.

Step #4: Evaluate

Last, we need to evaluate our trained network. For each of the images in our testing set, we present them to the network and ask it to *predict* what it thinks the label of the image is. We then tabulate the predictions of the model for an image in the testing set.

Finally, these *model predictions* are compared to the *ground-truth* labels from our testing set. The ground-truth labels represent what the image category *actually is*. From there, we can compute the number of predictions our classifier got correct and compute aggregate reports such as precision, recall, and f-measure, which are used to quantify the performance of our network as a whole.

Feature-based Learning versus Deep Learning for Image Classification

In the traditional, feature-based approach to image classification, there is actually a step inserted between Step #2 and Step #3 – this step is ***feature extraction***. During this phase, we apply hand-engineered algorithms such as HOG [32], LBPs [21], etc. to quantify the contents of an image based on a particular component of the image we want to encode (i.e., shape, color, texture). Given these features, we then proceed to train our classifier and evaluate it.

When building Convolutional Neural Networks, we can actually *skip* the feature extraction step. The reason for this is because CNNs are *end-to-end* models. We present the raw input data (pixels) to the network. The network then *learns* filters inside its hidden layers that can be used to discriminate amongst object classes. The output of the network is then a probability distribution over class labels.

One of the exciting aspects of using CNNs is that we no longer need to fuss over hand-engineered features – we can let our network learn the features instead. However, this tradeoff does come at a cost. Training CNNs can be a non-trivial process, so be prepared to spend considerable time familiarizing yourself with the experience and running many experiments to determine what does and does not work.

What Happens When my Predictions Are Incorrect?

Inevitably, you will train a deep learning network on your training set, evaluate it on your test set (finding that it obtains high accuracy), and then apply it to images that are *outside* both your training and testing set – *only to find that the network performs poorly*.

This problem is called **generalization**, the ability for a network to *generalize* and correctly predict the class label of an image that does not exist as part of its training or testing data.

The ability for a network to generalize is quite literally the most important aspect of deep learning research – if we can train networks that can generalize to outside datasets without re-training or fine-tuning, we'll make great strides in machine learning, enabling networks to be re-used in a variety of domains. The ability of a network to generalize will be discussed many times in this book, but I wanted to bring up the topic now since you will inevitably run into generalization issues, especially as you learn the ropes of deep learning.

Instead of becoming frustrated with your model not correctly classifying an image, consider the set of factors of variation mentioned above. Does your training dataset accurately reflect examples of these factors of variation? If not, you'll need to gather more training data (and read the rest of this book to learn other techniques to combat generalization).

Summary

Inside this chapter we learned what image classification is and why it's such a challenging task for computers to perform well on (even though humans do it intuitively with seemingly no effort). We then discussed the three main types of machine learning, supervised learning, unsupervised learning, semi-supervised learning – this book primarily focuses on supervised learning where we have *both* the training examples *and* the class labels associated with them. Semi-supervised learning and unsupervised learning are both open areas of research for deep learning (and in machine learning in general).

Finally, we reviewed the four steps in the deep learning classification pipeline. These steps including gathering your dataset, splitting your data into training, testing, and validation steps, training your network, and finally evaluating your model.

Unlike traditional feature-based approaches which require us to utilize hand-crafted algorithms to extract features from an image, image classification models, such as Convolutional Neural Networks, are end-to-end classifiers which internally learn features that can be used to discriminate amongst image classes.

5. Datasets for Image Classification

At this point we have accustomed ourselves to the fundamentals of the image classification pipeline – but before we dive into any code looking at actually *how* to take a dataset and build an image classifier, let's first review datasets that you'll see inside *Deep Learning for Computer Vision with Python*.

Some of these datasets are essentially "solved", enabling us to obtain extremely high-accuracy classifiers (> 95% accuracy) with little effort. Other datasets represent categories of computer vision and deep learning problems are still open research topics today and are far from solved. Finally, a few of the datasets are part of image classification competitions and challenges (e.x., Kaggle Dogs vs. Cats and cs231n Tiny ImageNet 200).

It's important to review these datasets now so that we have a high-level understanding of the challenges we can expect when working with them in later chapters.

MNIST

Figure 5.1: A sample of the MNIST dataset. The goal of this dataset is to correctly classify the handwritten digits, $0 - 9$.

The MNIST ("NIST" stands for *National Institute of Standards and Technology* while the "M" stands for "*modified*" as the data has been preprocessed to reduce any burden on computer vision processing and focus solely on the task of digit recognition) dataset is one of the most well studied datasets in the computer vision and machine learning literature.

The goal of this dataset is to correctly classify the handwritten digits $0 - 9$. In many cases, this dataset is a benchmark, a standard to which machine learning algorithms are ranked. In fact,

MNIST is *so well studied* that Geoffrey Hinton described the dataset as *"the drosophila of machine learning"* [10] (a *drosophila* is a genus of fruit fly), comparing how budding biology researchers use these fruit flies as they are easily cultured en masse, have a short generation time, and mutations are easily obtained.

In the same vein, the MNIST dataset is simple dataset for early deep learning practitioners to get their "first taste" of training a neural network without too much effort (it's *very* easy to obtain $> 97\%$ classification accuracy) – training a neural network model on MNIST is very much the *"Hello, World"* equivalent in machine learning.

MNIST itself consists of 60,000 training images and 10,000 testing images. Each feature vector is 784-dim, corresponding to the 28×28 grayscale pixel intensities of the image. These grayscale pixel intensities are unsigned integers, falling into the range $[0, 255]$. All digits are placed on a black background with the foreground being white and shades of gray. Given these raw pixel intensities, our goal is to train a neural network to correctly classify the digits.

We'll be primarily using this dataset in the early chapters of the *Starter Bundle* to help us "get our feet wet" and learn the ropes of neural networks.

Animals: Dogs, Cats, and Pandas

Figure 5.2: A sample of the 3-class animals dataset consisting of 1,000 images per dog, cat, and panda class respectively for a total of 3,000 images.

The purpose of this dataset is to correctly classify an image as contain a dog, cat, or panda. Containing only 3,000 images, the Animals dataset is meant to be another "introductory" dataset that we can quickly train a deep learning model on either our CPU or GPU and obtain reasonable accuracy.

In Chapter 10 we'll use this dataset to demonstrate how using the pixels of an image as a feature vector does not translate to a high-quality machine learning model *unless* we employ the usage of a Convolutional Neural Network (CNN).

Images for this dataset were gathered by sampling the Kaggle Dogs vs. Cats images along with the ImageNet dataset for panda examples. The Animals dataset is primarily used in the *Starter Bundle* only.

CIFAR-10

Figure 5.3: Example images from the ten class CIFAR-10 dataset.

Just like MNIST, CIFAR-10 is considered another standard benchmark dataset for image classification in the computer vision and machine learning literature. CIFAR-10 consists of 60,000 $32 \times 32 \times 3$ (RGB) images resulting in a feature vector dimensionality of 3072.

As the name suggests, CIFAR-10 consists of 10 classes, including: *airplanes*, *automobiles*, *birds*, *cats*, *deer*, *dogs*, *frogs*, *horses*, *ships*, and *trucks*.

While it's quite easy to train a model that obtains $> 97\%$ classification accuracy on MNIST, it's *substantially harder* obtain such a model for CIFAR-10 (and it's bigger brother, CIFAR-100) [50].

The challenge comes from the *dramatic variance* in how objects appear. For example, we can no longer assume that an image containing a green pixel at a given (x, y)-coordinate is a frog. This pixel could be part of the background of a forest that contains a deer. Or, the pixel could simply be the color of a green truck.

These assumptions are a stark contrast to the MNIST dataset where the network can learn assumptions regarding the spatial distribution of pixel intensities. For example, the spatial distribution of foreground pixels of the number 1 is substantially different than a 0 or 5.

While being a small dataset, CIFAR-10 is still regularly used to benchmark new CNN architectures. We'll be using CIFAR-10 in both the *Starter Bundle* and the *Practitioner Bundle*.

SMILES

As the name suggests, the SMILES dataset [51] consists of images of faces that are either *smiling* or *not smiling*. In total, there are 13,165 grayscale images in the dataset, with each image having a size of 64×64.

Images in this dataset are *tightly cropped* around the face allowing us to devise machine learning algorithms that focus solely on the task of smile recognition. Decoupling *computer vision preprocessing* from *machine learning* (especially for benchmark datasets) is a common trend you'll

Figure 5.4: Top: Examples of "smiling" faces. Bottom: Samples of "not smiling" faces. We will later train a Convolutional Neural Network to recognize between smiling and not smiling faces in real-time video streams.

see when reviewing popular benchmark datasets. In some cases, it's unfair to assume that a machine learning researcher has enough exposure to computer vision to properly preprocess a dataset of images prior to applying their own machine learning algorithms.

That said, this trend is *quickly changing*, and any practitioner interested in applying machine learning to computer vision problems is assumed to have at least a rudimentary background in computer vision. This trend will continue in the future, so if you plan on studying deep learning for computer vision in any depth, definitely be sure to supplement your education with a bit of computer vision, even if it's just the fundamentals.

If you find that you need to improve your computer vision skills, take a look at [8].

Kaggle: Dogs vs. Cats

The Dogs vs. Cats challenge is part of a Kaggle competition to devise a learning algorithm that can correctly classify an image as containing a *dog* or a *cat*. A total of 25,000 images are provided to train your algorithm with *varying image resolutions*. A sample of the dataset can be seen in Figure 5.5.

How you decide to preprocess your images can lead to varying performance levels, again demonstrating that a background in computer vision and image processing basics will go a long way when studying deep learning.

We'll be using this dataset in the *Practitioner Bundle* when I demonstrate how to claim a top-25 position on the Kaggle Dogs vs. Cats leaderboard using the AlexNet architecture.

Flowers-17

The Flowers-17 dataset is a 17 category dataset with 80 images per class curated by Nilsback et al. [52]. The goal of this dataset is to correctly predict the species of flower for a given input image. A sample of the Flowers-17 dataset can be seen in Figure 5.6.

Flowers-17 can be considered a challenging dataset due to the dramatic changes in scale, viewpoint angles, background clutter, varying lighting conditions, and intra-class variation. Furthermore, with only 80 images per class, it becomes challenging for deep learning models to learn a representation for each class *without* overfitting. As a general rule of thumb, it's advisable to have 1,000-5,000 example images *per class* when training a deep neural network [10].

Figure 5.5: Samples from the Kaggle Dogs vs. Cats competition. The goal of this 2-class challenge is to correctly identify a given input image as containing a "dog" or "cat".

We will study the Flowers-17 dataset inside the *Practitioner Bundle* and explore methods to improve classification using transfer learning methods such as feature extraction and fine-tuning.

CALTECH-101

Introduced by Fei-Fei et al. [53] in 2004, the CALTECH-101 dataset is a popular benchmark dataset for object detection. Typically used for *object detection* (i.e., predicting the (x, y)-coordinates of the bounding box for a particular object in an image), we can use CALTECH-101 to study deep learning algorithms as well.

The dataset of 8,677 images includes 101 categories spanning a diverse range of objects, including elephants, bicycles, soccer balls, and even human brains, just to name a few. The CALTECH-101 dataset exhibits *heavy* class imbalances (meaning that there are more example images for some categories than others), making it interesting to study from a class imbalance perspective.

Previous approaches to classifying images to CALTECH-101 obtained accuracies in the range of 35-65% [54, 55, 56]. However, as I'll demonstrate in the *Practitioner Bundle*, **it's easy for us to leverage deep learning for image classification to obtain over 99% classification accuracy**.

Tiny ImageNet 200

Stanford's excellent *cs231n: Convolutional Neural Networks for Visual Recognition* class [57] has put together an image classification challenge for students similar to the ImageNet challenge, but smaller in scope. There are a total of 200 image classes in this dataset with 500 images for training, 50 images for validation, and 50 images for testing per class. Each image has been preprocessed and cropped to $64 \times 64 \times 3$ pixels making it easier for students to focus on deep learning techniques rather than computer vision preprocessing functions.

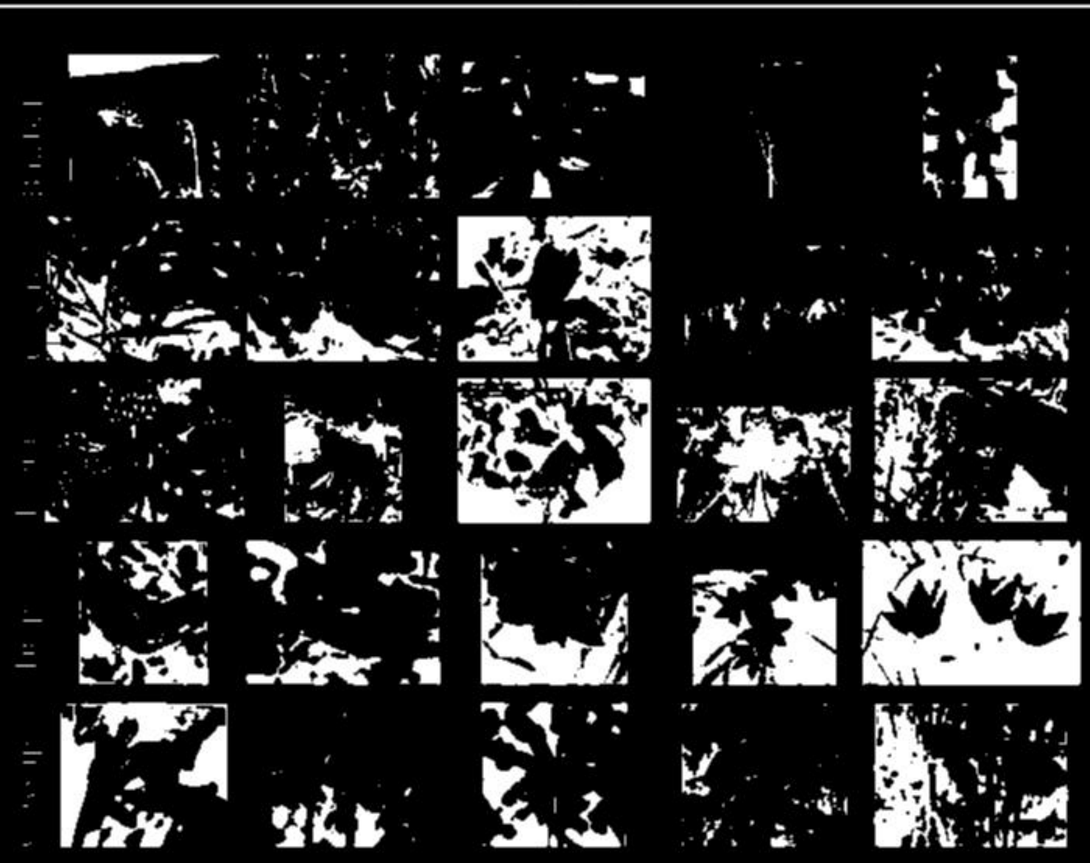

Figure 5.6: A sample of five (out of the seventeen total) classes in the Flowers-17 dataset where each class represents a *specific* flower species.

However, as we'll find in the *Practitioner Bundle*, the preprocessing steps applied by Karpathy and Johnson actually make the problem a bit *harder* as some of the important, discriminating information is cropped out during the preprocessing task. That said, I'll be demonstrating how to train the VGGNet, GoogLenet, and ResNet architectures on this dataset and claim a top position on the leaderboard.

Adience

The Adience dataset, constructed by Eidinger et al. 2014 [58], is used to facilitate the study of age and gender recognition. A total of 26,580 images are included in the dataset with ages ranging from 0-60. The goal of this dataset is to correctly predict *both* the age and gender of the subject in the image. We'll discuss the Adience dataset further (and build our own age and gender recognition systems) inside the *ImageNet Bundle*. You can see a sample of the Adience dataset in Figure 5.7.

ImageNet

Within the computer vision and deep learning communities, you might run into a bit of contextual confusion regarding what ImageNet is and isn't.

What Is ImageNet?

ImageNet is actually a *project* aimed at labeling and categorizing images into almost 22,000 categories based on a defined set of words and phrases.

At the time of this writing, there are over *14 million images* in the ImageNet project. To organize such a massive amount of data, ImageNet follows the WordNet hierarchy [59]. Each meaningful word/phrase inside WordNet is called a "synonym set" or "synset" for short. Within the ImageNet project, images are organized according to these synsets, with the goal being to have 1,000+ images per synset.

ImageNet Large Scale Visual Recognition Challenge (ILSVRC)

In the context of computer vision and deep learning, whenever you hear people talking about ImageNet, they are very likely referring to the *ImageNet Large Scale Visual Recognition Challenge* [42],

Contents

1.1 I Studied Deep Learning the *Wrong Way*... This Is the *Right Way* 15
1.2 Who This Book Is For 17
1.2.1 Just Getting Started in Deep Learning? 17
1.2.2 Already a Seasoned Deep Learning Practitioner? 17
1.3 Book Organization 17
1.3.1 Volume #1: Starter Bundle 17
1.3.2 Volume #2: Practitioner Bundle 18
1.3.3 Volume #3: ImageNet Bundle 18
1.3.4 Need to Upgrade Your Bundle? 18
1.4 Tools of the Trade: Python, Keras, and Mxnet 18
1.4.1 What About TensorFlow? 18
1.4.2 Do I Need to Know OpenCV? 19
1.5 Developing Our Own Deep Learning Toolset 19
1.6 Summary 20

2.1 A Concise History of Neural Networks and Deep Learning 22
2.2 Hierarchical Feature Learning 24
2.3 How "Deep" Is Deep? 27
2.4 Summary 30

3.1 Pixels: The Building Blocks of Images 31
3.1.1 Forming an Image From Channels 34

Figure 5.7: A sample of the Adience dataset for age and gender recognition. Age ranges span from 0-60+.

or simply ILSVRC for short.

The goal of the image classification track in this challenge is to train a model that can classify an image into *1,000 separate categories* using approximately 1.2 million images for training, 50,000 for validation, and 100,000 for testing. 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. You can find the full list of object categories in the ILSVRC challenge here: .

When it comes to image classification, the ImageNet challenge is the *de facto* standard for computer vision classification algorithms – and the leaderboard for this challenge has been *dominated* by Convolutional Neural Networks and deep learning techniques since 2012.

Inside the *ImageNet Bundle* I'll be demonstrating how to train seminal network architectures (AlexNet, SqueezeNet, VGGNet, GoogLeNet, ResNet) *from scratch* on this popular dataset, allowing you to replicate the state-of-the-art results you see in the respective research papers.

Kaggle: Facial Expression Recognition Challenge

Another challenge put together by Kaggle, the goal of the Facial Expression Recognition Challenge (FER) is to correctly identify the *emotion* a person is experiencing simply from a picture of their face. A total of 35,888 images are provided in the FER challenge with the goal to label a given facial expression into seven different categories:

1. Angry
2. Disgust (sometimes grouped in with "Fear" due to class imbalance)
3. Fear
4. Happy
5. Sad
6. Surprise

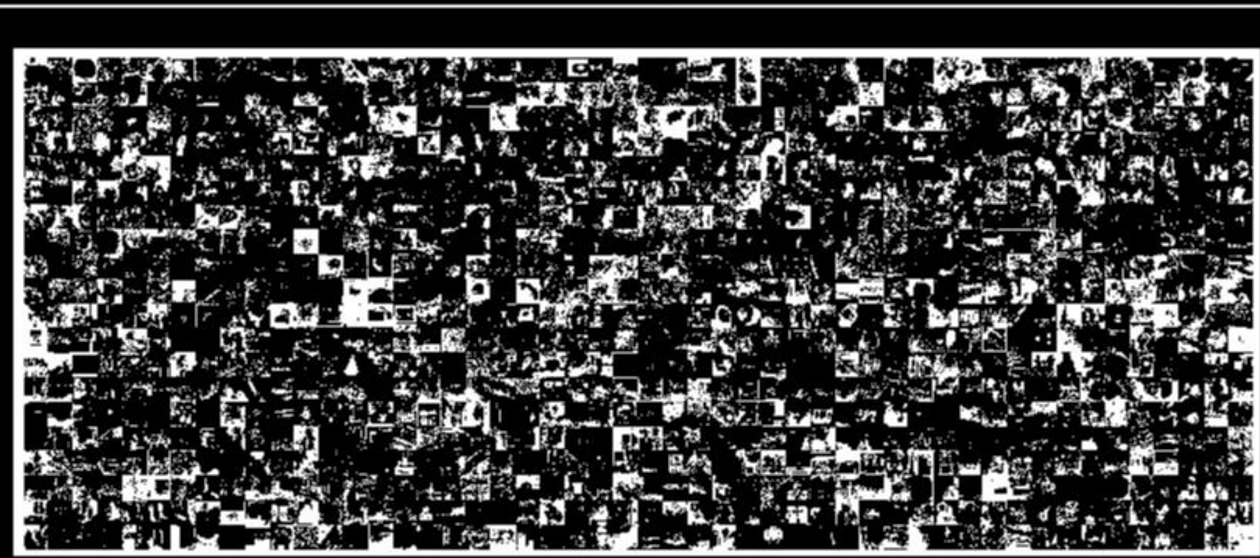

Figure 5.8: A collage of ImageNet examples put together by Stanford University. This dataset is massive with over 1.2 million images and 1,000 possible object categories. ImageNet is considered the *de facto* standard for benchmarking image classification algorithms.

7. Neutral

I'll be demonstrating how to use this dataset for emotion recognition inside the *ImageNet Bundle*.

Indoor CVPR

The Indoor Scene Recognition dataset [60], as the name suggests, consists of a number of indoor scenes, including stores, houses, leisure spaces, working areas, and public spaces. The goal of this dataset is to correctly train a model that can recognize each of the areas. However, instead of using this dataset for its original intended purpose, we'll instead be using it inside the *ImageNet Bundle* to automatically **detect** and **correct** image orientation.

Stanford Cars

Another dataset put together by Stanford, the Cars Dataset [61] consists of 16,185 images of 196 classes of cars. You can slice-and-dice this dataset any way you wish based on vehicle make, model, or even manufacturer year. Even though there are relatively few images per class (with heavily class imbalances), I'll demonstrate how to use Convolutional Neural Networks to obtain $> 95\%$ **classification accuracy** when labeling the make and model of a vehicle.

Summary

In this chapter, we reviewed the datasets you'll encounter in the remainder of *Deep Learning for Computer Vision with Python*. Some of these datasets are considered "toy" datasets, small sets of images that we can use to learn the ropes of neural networks and deep learning. Other datasets are popular due to historical reasons and serve as excellent benchmarks to evaluate new model architectures. Finally, datasets such as ImageNet are still open-ended research topics and are used to advance the state-of-the-art for deep learning.

Take the time to briefly familiarize yourself with these datasets now – I'll be discussing each of the datasets in detail when they are first introduced in their respective chapters.

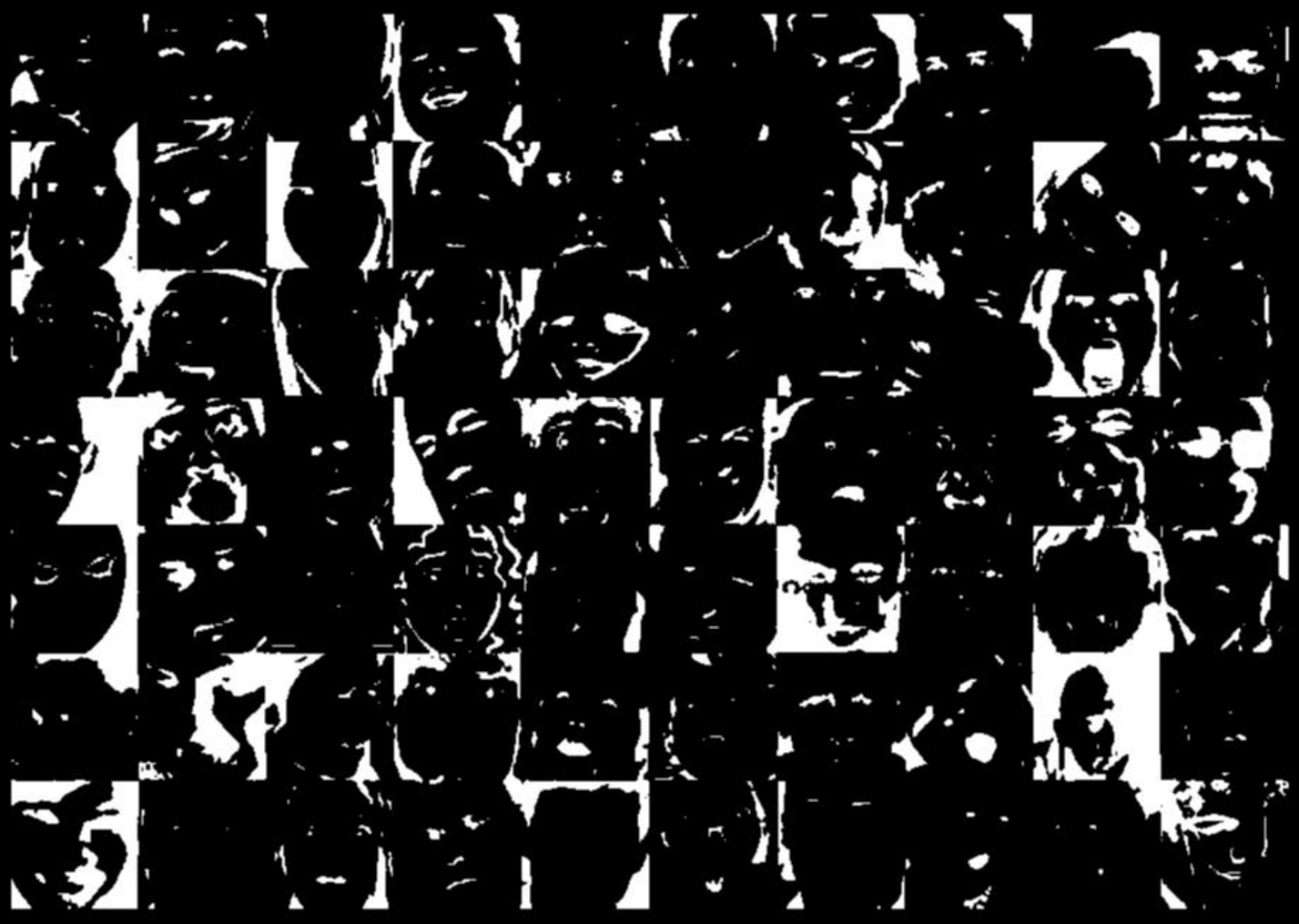

Figure 5.9: A sample of the facial expressions inside the Kaggle: Facial Expression Recognition Challenge. We will train a CNN to recognize and identify each of these emotions. This CNN will also be able to run in *real-time* on your CPU, enabling you to recognize emotions in video streams.

Figure 5.10: The Stanford Cars Dataset consists of 16,185 images with 196 vehicle make and model classes. We'll learn how obtain $> 95\%$ classification accuracy on this dataset inside the *ImageNet Bundle*.

6. Configuring Your Development Environment

When it comes to learning a new technology (especially deep learning), configuring your development environment tends to be half the battle. Between different operating systems, varying dependency versions, and the actual libraries themselves, configuring your own deep learning development environment can be quite the headache.

These issues are all *further compounded* by the speed in which deep learning libraries are updated and released – new features push innovation, but also break previous versions. The CUDA Toolkit, in particular, is a great example: on average there are 2-3 *new releases* of CUDA *every year*.

With each new release brings optimizations, new features, and the ability to train neural networks faster. But each release further complicates backward compatibility. This fast release cycle implies that deep learning is not only dependent on *how* you configured your development environment but *when* you configured it as well. **Depending on the timeframe, *your environment may be obsolete!***

Due to the rapidly changing nature of deep learning dependencies and libraries, I've decided to move much of this chapter to the **Companion Website** ( ) so that new, fresh tutorials will *always* be available for you to use.

You should use this chapter to help familiarize yourself with the various deep learning libraries we'll be using in this book, then follow instructions on the pages that link to those libraries from this book.

Libraries and Packages

In order to become a successful deep learning practitioner, we need the right set of tools and packages. This section details the programming language along with the primary libraries we'll be using to study deep learning for computer vision.

Python

We'll be utilizing the Python programming language for all examples inside *Deep Learning for Computer Vision with Python*. Python is an easy language to learn and is hands-down the **best**

way to work with deep learning algorithms. The simple, intuitive syntax allows you to focus on learning the basics of deep learning, rather than spending hours fixing crazy compiler errors in other languages.

Keras

To build and train our deep learning networks we'll primarily be using the Keras library. Keras supports *both* TensorFlow and Theano, making it *super easy* to build and train networks quickly. Please refer to Section 6.2 for more information on TensorFlow and Theano compatibility with Keras.

Mxnet

We'll also be using mxnet, a deep learning library that specializes in *distributed, multi-machine learning*. The ability to parallelize training across multiple GPUs/devices is *critical* when training deep neural network architectures on massive image datasets (such as ImageNet).

> The mxnet library is only used in the *ImageNet Bundle* of this book.

OpenCV, scikit-image, scikit-learn, and more

Since this book focuses on applying deep learning to *computer vision*, we'll be leveraging a few extra libraries as well. You *do not* need to be an expert in these libraries or have prior experience with them to be successful when using this book, but I *do* suggest familiarizing yourself with the basics of OpenCV if you can. The first five chapters of are more than sufficient to understand the basics of the OpenCV library.

The main goal of OpenCV is real-time image processing. This library has been around since 1999, but it wasn't until the 2.0 release in 2009 that we saw the incredible Python support which included representing images as NumPy arrays.

OpenCV itself is written in C/C++, but Python bindings are provided when running the install. OpenCV is *hands down* the *de-facto standard* when it comes to image processing, so we'll make use of it when loading images from disk, displaying them to our screen, and performing basic image processing operations.

To complement OpenCV, we'll also be using a tiny bit of scikit-image [62] ( ), a collection of algorithms for image processing.

Scikit-learn [5] ( ), is an open-source Python library for machine learning, cross-validation, and visualization – this library complements Keras well and helps us from not "reinventing the wheel", especially when it comes to creating training/testing/validation splits and validating the accuracy of our deep learning models.

Configuring Your Development Environment?

If you're ready to configure your deep learning environment, just click the link below and follow the provided instructions for your operating system and whether or not you will be using a GPU:

> If you have not already created your account on the companion website for *Deep Learning for Computer Vision with Python*, please see the first few pages of this book (immediately following the Table of Contents) for the registration link. From there, create your account and you'll be able to access the supplementary material.

Preconfigured Virtual Machine

I realize that configuring your development environment can not only be a *time consuming, tedious task*, but potentially a *major barrier to entry* if you're new to Unix-based environments. Because of this difficulty, your purchase of *Deep Learning for Computer Vision with Python* includes a preconfigured Ubuntu VirtualBox virtual machine that ships with all the necessary deep learning and computer vision libraries you'll need to be successful when using this book *preconfigured* and *pre-installed*.

Make sure you download the `VirtualMachine.zip` included with your bundle to have access to this virtual machine. Instructions on how to setup and use your virtual machine can be found inside the `README.pdf` included with your download of this book.

Cloud-based Instances

A major downside of the Ubuntu VM is that by the very definition of a virtual machine, a VM is not allowed to access the physical components of your host machine (such as a GPU). When training larger deep learning networks, having a GPU is extremely beneficial.

For those who wish to have access to a GPU when training their neural networks, I would suggest either:

1. Configuring an Amazon EC2 instance with GPU support.
2. Signing up for a FloydHub account and configure your GPU instance in the cloud.

It's important to note that each of these options charge based on the number of hours (EC2) or seconds (FloydHub) that your instance is booted for. If you decide to go the "GPU in the cloud route" be sure to compare prices and be conscious of your spending – there is nothing worse than getting a large, unexpected bill for cloud usage.

If you choose to use a cloud-based instance then I would encourage you to use my pre-configured Amazon Machine Instance (AMI). The AMI comes with all deep learning libraries you'll need in this book pre-configured and pre-installed.

To learn more about the AMI, please refer to *Deep Learning for Computer Vision with Python* companion website.

How to Structure Your Projects

Now that you've had a chance to configure your development environment, take a second now and download the `.zip` of the code and datasets associated with the *Starter Bundle*.

After you've downloaded the file, unarchive it, and you'll see the following directory structure:

```
|--- sb_code
|    |--- chapter07-first_image_classifier
|    |--- chapter08-parameterized_learning
|    |--- chapter09-optimization_methods
...
|    |--- datasets
```

Each chapter (that includes accompanying code) has its own directory. Each directory then includes:

- The source code for the chapter.
- The `pyimagesearch` library for deep learning that you'll be creating as you follow along with the book.
- Any additional files needed to run the respective examples.

The `datasets` directory, as the name implies, contains all image datasets for the *Starter Bundle*.

As an example, let's say I wanted to train my first image classifier. I would first change directory into `chapter07-first_image_classifier` and then execute the `knn.py` script, pointing the `--dataset` command line argument to the `animals` dataset:

```
$ cd ../chapter07-first_image_classifier/
$ python knn.py --dataset ../datasets/animals
```

This will instruct the `knn.py` script to train a simple k-Nearest Neighbor (k-NN) classifier on the "animals" dataset (which is a subdirectory inside `datasets`), a small collection of dogs, cats, and pandas in images.

If you are new to the command line and how to use command line arguments, I ***highly recommend*** that you read up on command line arguments and how to use them before getting to far in this book:

Becoming comfortable with the command line (and how to debug errors using the terminal) is a very important skill for you to develop.

Finally, as a quick note, I wanted to mention that I prefer keeping my datasets *separate* from my source code as it:

- Keeps my project structure neat and tidy
- Allows me to reuse datasets across multiple projects

I would encourage you to adopt a similar directory structure for your own projects.

Summary

When it comes to configuring your deep learning development environment, you have a number of options. If you would prefer to work from your local machine, that's totally reasonable, but you will need to compile and install some dependencies first. If you are planning on using your CUDA-compatible GPU on your local machine, a few extra install steps will be required as well.

For readers who are new to configuring their development environment or for readers who simply want to skip the process altogether, be sure to take a look at the preconfigured Ubuntu VirtualBox virtual machine included in the download of your *Deep Learning for Computer Vision with Python* bundle.

If you would like to use a GPU but do not have one attached to your system, consider using cloud-based instances such as Amazon EC2 or FloydHub. While these services *do* incur an hourly charge for usage, they can save you money when compared to buying a GPU upfront.

Finally, please keep in mind that if you plan on doing any serious deep learning research or development, consider using a Linux environment such as Ubuntu. While deep learning work can *absolutely* be done on Windows (not recommended) or macOS (totally acceptable if you are just getting started), nearly all production-level environments for deep learning leverage Linux-based operating systems – keep this fact in mind when you are configuring your own deep learning development environment.

7. Your First Image Classifier

Over the past few chapters we've spent a reasonable amount of time discussing image fundamentals, types of learning, and even a four step pipeline we can follow when building our own image classifiers. But we have yet to build an *actual* image classifier of our own.

That's going to change in this chapter. We'll start by building a few helper utilities to facilitate preprocessing and loading images from disk. From there, we'll discuss the k-Nearest Neighbors (k-NN) classifier, your first exposure to using machine learning for image classification. In fact, this algorithm is *so simple* that it doesn't do any actual "learning" at all – yet it is still an important algorithm to review so we can appreciate how neural networks learn from data in future chapters.

Finally, we'll apply our k-NN algorithm to recognize various species of animals in images.

Working with Image Datasets

When working with image datasets, we first must consider the total size of the dataset in terms of bytes. Is our dataset large enough to fit into the available RAM on our machine? Can we load the dataset as if loading a large matrix or array? Or is the dataset so large that it exceeds our machine's memory, requiring us to "chunk" the dataset into segments and only load parts at a time?

The datasets inside the *Starter Bundle* are all small enough that we can load them into main memory without having to worry about memory management; however, the much larger datasets inside the *Practitioner Bundle* and *ImageNet Bundle* will require us to develop some clever methods to efficiently handle loading images in a way that we can train an image classifier (without running out of memory).

That said, you should always be cognizant of your dataset size before even starting to work with image classification algorithms. As we'll see throughout the rest of this chapter, taking the time to organize, preprocess, and load your dataset is a critical aspect of building an image classifier.

Introducing the "Animals" Dataset

The "Animals" dataset is a simple example dataset I put together to demonstrate how to train image classifiers using simple machine learning techniques as well as advanced deep learning algorithms.

Figure 7.1: A sample of the 3-class animals dataset consisting of 1,000 images per *dog*, *cat*, and *panda* class respectively for a total of 3,000 images.

Images inside the Animals dataset belong to three distinct classes: **dogs**, **cats**, and **pandas**, with 1,000 example images per class. The dog and cat images were sampled from the Kaggle Dogs vs. Cats challenge ( ) while the panda images were sampled from the ImageNet dataset [42].

Containing only 3,000 images, the Animals dataset can easily fit into the main memory of our machines, which will make training our models much faster, without requiring us to write any "overhead code" to manage a dataset that could not otherwise fit into memory. Best of all, a deep learning model can quickly be trained on this dataset on *either* a CPU or GPU. Regardless of your hardware setup, you can use this dataset to learn the basics of machine learning and deep learning.

Our goal in this chapter is to leverage the k-NN classifier to attempt to recognize each of these species in an image using only the *raw pixel intensities* (i.e., no feature extraction is taking place). As we'll see, raw pixel intensities do not lend themselves well to the k-NN algorithm. Nonetheless, this is an important benchmark experiment to run so we can appreciate why Convolutional Neural Networks are able to obtain such high accuracy on raw pixel intensities while traditional machine learning algorithms fail to do so.

The Start to Our Deep Learning Toolkit

As I mentioned in Section 1.5, we'll be building our own custom deep learning toolkit throughout the entirety of this book. We'll start with basic helper functions and classes to preprocess images and load small datasets, eventually building up to implementations of current state-of-the-art Convolutional Neural Networks.

In fact, this is the *exact same* toolkit I use when performing deep learning experiments of my own. This toolkit will be built piece by piece, chapter by chapter, allowing you to see the individual components that make up the package, eventually becoming a full-fledged library that can be used to rapidly build and train your own custom deep learning networks.

Let's go ahead and start defining the project structure of our toolkit:

3.2 The Image Coordinate System 34
3.2.1 Images as NumPy Arrays 35
3.2.2 RGB and BGR Ordering 36
3.3 Scaling and Aspect Ratios 36
3.4 Summary 38

4.1 What Is Image Classification? 40
4.1.1 A Note on Terminology 40
4.1.2 The Semantic Gap 41
4.1.3 Challenges 42
4.2 Types of Learning 45
4.2.1 Supervised Learning 45
4.2.2 Unsupervised Learning 46
4.2.3 Semi-supervised Learning 47
4.3 The Deep Learning Classification Pipeline 48
4.3.1 A Shift in Mindset 48
4.3.2 Step #1: Gather Your Dataset 50
4.3.3 Step #2: Split Your Dataset 50
4.3.4 Step #3: Train Your Network 51
4.3.5 Step #4: Evaluate 51
4.3.6 Feature-based Learning versus Deep Learning for Image Classification 51
4.3.7 What Happens When my Predictions Are Incorrect? 52
4.4 Summary 52

5.1 MNIST 53
5.2 Animals: Dogs, Cats, and Pandas 54
5.3 CIFAR-10 55
5.4 SMILES 55
5.5 Kaggle: Dogs vs. Cats 56
5.6 Flowers-17 56
5.7 CALTECH-101 57
5.8 Tiny ImageNet 200 57
5.9 Adience 58
5.10 ImageNet 58
5.10.1 What Is ImageNet? 58
5.10.2 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 58
5.11 Kaggle: Facial Expression Recognition Challenge 59
5.12 Indoor CVPR 60
5.13 Stanford Cars 60
5.14 Summary 60

```
|--- pyimagesearch
```

As you can see, we have a single module named `pyimagesearch`. All code that we develop will exist inside the `pyimagesearch` module. For the purposes of this chapter, we'll need to define two submodules:

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

The `datasets` submodule will start our implementation of a class named `SimpleDatasetLoader`. We'll be using this class to load small image datasets from disk (that can fit into main memory), optionally preprocess each image in the dataset according to a set of functions, and then return the:

1. Images (i.e., raw pixel intensities)
2. Class label associated with each image

We then have the `preprocessing` submodule. As we'll see in later chapters, there are a number of preprocessing methods we can apply to our dataset of images to boost classification accuracy, including mean subtraction, sampling random patches, or simply resizing the image to a fixed size. In this case, our `SimplePreprocessor` class will do the latter – load an image from disk and resized it to a fixed size, ignoring aspect ratio. In the next two sections we'll implement `SimplePreprocessor` and `SimpleDatasetLoader` by hand.

> While we will be reviewing the entire `pyimagesearch` module for deep learning in this book, I have purposely left explanations of the `__init__.py` files as an exercise to the reader. These files simply contain shortcut imports and are not relevant to understanding the deep learning and machine learning techniques applied to image classification. If you are new to the Python programming language, I would suggest brushing up on the basics of package imports [63] ( ).

A Basic Image Preprocessor

Machine learning algorithms such as k-NN, SVMs, and even Convolutional Neural Networks require all images in a dataset to have a fixed feature vector size. In the case of images, this requirement implies that our images must be preprocessed and scaled to have identical widths and heights.

There are a number of ways to accomplish this resizing and scaling, ranging from more advanced methods that respect the aspect ratio of the original image to the scaled image to simple methods that ignore the aspect ratio and simply squash the width and height to the required dimensions. Exactly which method you should use really depends on the complexity of your *factors of variation* (Section 4.1.3) – in some cases, ignoring the aspect ratio works just fine; in other cases, you'll want to preserve the aspect ratio.

In this chapter, we'll start with the basic solution: building an image preprocessor that resizes the image, ignoring the aspect ratio. Open up `simplepreprocessor.py` and then insert the following code:

```
1
2         cv2
3
4       SimplePreprocessor:
5                 (    , width, height, inter=cv2.INTER_AREA):
6
7
8                 .width = width
9                 .height = height
10                .inter = inter
11
12                    (    , image):
13
14
15                  cv2.resize(image, (    .width,     .height),
16                interpolation=    .inter)
```

Line 2 imports our only required package, our OpenCV bindings. We then define the constructor to the `SimpleProcessor` class on **Line 5**. The constructor requires two arguments, followed by a third optional one, each detailed below:

- `width`: The target width of our input image after resizing.
- `height`: The target height of our input image after resizing.
- `inter`: An optional parameter used to control which interpolation algorithm is used when resizing.

The `preprocess` function is defined on **Line 12** requiring a single argument – the input `image` that we want to preprocess.

Lines 15 and 16 preprocess the image by resizing it to a fixed size of `width` and `height` which we then return to the calling function.

Again, this preprocessor is by definition very basic – all we are doing is accepting an input image, resizing it to a fixed dimension, and then returning it. However, when combined with the image dataset loader in the next section, this preprocessor will allow us to quickly load and preprocess a dataset from disk, enabling us to briskly move through our image classification pipeline and move onto more important aspects, *such as training our actual classifier*.

Building an Image Loader

Now that our `SimplePreprocessor` is defined, let's move on to the `SimpleDatasetLoader`:

```
1
2         numpy     np
3         cv2
4         os
5
6       SimpleDatasetLoader:
7                 (    , preprocessors=    ):
8
9                 .preprocessors = preprocessors
10
11
12
13                .preprocessors        :
14                  .preprocessors = []
```

Lines 2-4 import our required Python packages: NumPy for numerical processing, `cv2` for our OpenCV bindings, and `os` so we can extract the names of subdirectories in image paths.

Line 7 defines the constructor to `SimpleDatasetLoader` where we can optionally pass in a list of image preprocessors (such as `SimpleProcessor`) that can be sequentially applied to a given input image.

Specifying these `preprocessors` as a list rather than a single value is important – there will be times where we first need to resize an image to a fixed size, then perform some sort of scaling (such as mean subtraction), followed by converting the image array to a format suitable for Keras. Each of these preprocessors can be implemented *independently*, allowing us to apply them sequentially to an image in an efficient manner.

We can then move on to the `load` method, the core of the `SimpleDatasetLoader`:

```
16                  (     , imagePaths, verbose=- ):
17
18              data = []
19              labels = []
20
21
22                  (i, imagePath)              (imagePaths):
23
24
25
26                   image = cv2.imread(imagePath)
27                   label = imagePath.split(os.path.sep)[- ]
```

Our `load` method requires a single parameter – `imagePaths`, which is a list specifying the file paths to the images in our dataset residing on disk. We can also supply a value for `verbose`. This "verbosity level" can be used to print updates to a console, allowing us to monitor how many images the `SimpleDatasetLoader` has processed.

Lines 18 and 19 initialize our `data` list (i.e., the images themselves) along with `labels`, the list of class labels for our images.

On **Line 22** we start looping over each of the input images. For each of these images, we load it from disk (**Line 26**) and extract the class label based on the file path (**Line 27**). We make the assumption that our datasets are organized on disk according to the following directory structure:

```
/dataset_name/class/image.jpg
```

The `dataset_name` can be whatever the name of the dataset is, in this case `animals`. The `class` should be the name of the class label. For our example, `class` is either `dog`, `cat`, or `panda`. Finally, `image.jpg` is the name of the actual image itself.

Based on this hierarchical directory structure, we can keep our datasets neat and organized. It is thus safe to assume that all images inside the `dog` subdirectory are examples of dogs. Similarly, we assume that all images in the `panda` directory contain examples of pandas.

Nearly every dataset that we review inside *Deep Learning for Computer Vision with Python* will follow this hierarchical directory design structure – **I *strongly encourage you* to do the same for your own projects as well.**

Now that our image is loaded from disk, we can preprocess it (if necessary):

```
29
30                          .preprocessors            :
```

```
31
32
33                      p        .preprocessors:
34                       image = p.preprocess(image)
35
36
37
38                data.append(image)
39                labels.append(label)
```

Line 30 makes a quick check to ensure that our `preprocessors` is not `None`. If the check passes, we loop over each of the `preprocessors` on **Line 33** and sequentially apply them to the image on **Line 34** – this action allows us to form a *chain of preprocessors* that can be applied to every image in a dataset.

Once the image has been preprocessed, we update the `data` and `label` lists, respectively (**Lines 39 and 39**).

Our last code block simply handles printing updates to our console and then returning a 2-tuple of the `data` and `labels` to the calling function:

```
41
42                  verbose >       i >       (i +  ) % verbose ==  :
43                         ("[INFO] processed {}/{}".format(i +  ,
44                            (imagePaths)))
45
46
47              (np.array(data), np.array(labels))
```

As you can see, our dataset loader is simple by design; however, it affords us the ability to apply any number of image processors to *every* image in our dataset with ease. The only caveat of this dataset loader is that it assumes that all images in the dataset can fit into main memory at once.

For datasets that are too large to fit into your system's RAM, we'll need to design a more complex dataset loader – I cover these more advanced dataset loaders inside the *Practitioner Bundle*. Now that we understand how to (1) preprocess an image and (2) load a collection of images from disk, we can now move on to the image classification stage.

k-NN: A Simple Classifier

The k-Nearest Neighbor classifier is *by far* the most simple machine learning and image classification algorithm. In fact, it's *so simple* that it doesn't actually "learn" anything. Instead, this algorithm directly relies on the distance between feature vectors (which in our case, are the raw RGB pixel intensities of the images).

Simply put, the k-NN algorithm classifies unknown data points by finding the *most common class* among the k *closest examples*. Each data point in the k closest data points casts a vote, and the category with the highest number of votes wins. Or, in plain English: *"Tell me who your neighbors are, and I'll tell you who you are"* [64], as Figure 7.2 demonstrates.

In order for the k-NN algorithm to work, it makes the primary assumption that images with similar visual contents lie *close together* in an n-dimensional space. Here, we can see three categories of images, denoted as *dogs*, *cats*, and *pandas*, respectively. In this pretend example we have plotted the "fluffiness" of the animal's coat along the $x-axis$ and the "lightness" of the coat along the y-axis. Each of the animal data points are grouped relatively close together in our

Figure 7.2: Given our dataset of dogs, cats, pandas, how might we classify the image outlined in red?

n-dimensional space. This implies that the distance between two *cat* images is *much smaller* than the distance between a *cat* and a *dog*.

However, in order to apply the k-NN classifier, we first need to select a distance metric or similarity function. A common choice includes the Euclidean distance (often called the L2-distance):

$$d(\boldsymbol{p}, \boldsymbol{q}) = \sqrt{\sum_{i=1}^{N} (q_i - p_i)^2} \tag{7.1}$$

However, other distance metrics such as the Manhattan/city block (often called the L1-distance) can be used as well:

$$d(\boldsymbol{p}, \boldsymbol{q}) = \sum_{i=1}^{N} |q_i - p_i| \tag{7.2}$$

In reality, you can use whichever distance metric/similarity function that most suits your data (and gives you the best classification results). However, for the remainder of this lesson, we'll be using the most popular distance metric: the Euclidean distance.

A Worked k-NN Example

At this point, we understand the principles of the k-NN algorithm. We know that it relies on the distance between feature vectors/images to make a classification. And we know that it requires a distance/similarity function to compute these distances.

But how do we actually *make* a classification? To answer this question, let's look at Figure 7.3. Here we have a dataset of three types of animals – *dogs*, *cats*, and *pandas* – and we have plotted them according to their *fluffiness* and *lightness of their coat*.

We have also inserted an "unknown animal" that we are trying to classify using only a **single neighbor** (i.e., $k = 1$). In this case, the nearest animal to the input image is a dog data point; thus our input image should be classified as *dog*.

Figure 7.3: In this example we have inserted an unknown image (highlighted in red) into the dataset and then used the distance between the unknown animal and dataset of animals to make the classification.

Let's try another "unknown animal", this time using $k = 3$ (Figure 7.4). We have found *two cats* and *one panda* in the *top three results*. Since the *cat* category has the largest number of votes, we'll classify our input image as *cat*.

We can keep performing this process for varying values of k, but no matter how large or small k becomes, the principle remains the same – the category with the largest number of votes in the k closest training points wins and is used as the label for the input data point.

In the event if a tie, the k-NN algorithm chooses one of the tied class labels at random.

Figure 7.4: Classifying another animal, only this time we used $k = 3$ rather than just $k = 1$. Since there are two cat images closer to the input image than the single panda image, we'll label this input input image as *cat*.

k-NN Hyperparameters

There are two clear hyperparameters that we are concerned with when running the k-NN algorithm. The first is obvious: the value of k. What is the optimal value of k? If it's too small (such as $k = 1$), then we gain efficiency but become susceptible to noise and outlier data points. However, if k is too large, then we are at risk of *over-smoothing* our classification results and increasing bias.

The second parameter we should consider is the actual distance metric. Is the Euclidean distance the best choice? What about the Manhattan distance?

In the next section, we'll train our k-NN classifier on the Animals dataset and evaluate the model on our testing set. I would encourage you to play around with different values of k along with varying distance metrics, noting how performance changes.

For an exhaustive review of how to tune k-NN hyperparameters, please refer to Lesson 4.3 inside the [33].

Implementing k-NN

The goal of this section is to train a k-NN classifier on the *raw pixel intensities* of the Animals dataset and use it to classify unknown animal images. We'll be using our four step pipeline to train classifiers from Section 4.3.2:

- **Step #1 – Gather Our Dataset:** The Animals datasets consists of 3,000 images with 1,000 images per dog, cat, and panda class, respectively. Each image is represented in the RGB

color space. We will preprocess each image by resizing it to 32×32 pixels. Taking into account the three RGB channels, the resized image dimensions imply that each image in the dataset is represented by $32 \times 32 \times 3 = 3,072$ integers.

- **Step #2 – Split the Dataset:** For this simple example, we'll be using *two* splits of the data. One split for training, and the other for testing. We will leave out the validation set for hyperparameter tuning and leave this as an exercise to the reader.
- **Step #3 – Train the Classifier:** Our k-NN classifier will be trained on the raw pixel intensities of the images in the training set.
- **Step #4 – Evaluate:** Once our k-NN classifier is trained, we can evaluate performance on the test set.

Let's go ahead and get started. Open up a new file, name it `knn.py`, and insert the following code:

```
1
2       sklearn.neighbors        KNeighborsClassifier
3       sklearn.preprocessing        LabelEncoder
4       sklearn.model_selection        train_test_split
5       sklearn.metrics        classification_report
6       pyimagesearch.preprocessing        SimplePreprocessor
7       pyimagesearch.datasets        SimpleDatasetLoader
8       imutils        paths
9         argparse
```

Lines 2-9 import our required Python packages. The most important ones to take note of are:

- **Line 2**: The `KNeighborsClassifier` is our implementation of the k-NN algorithm, provided by the scikit-learn library.
- **Line 3:** `LabelEncoder`, a helper utility to convert labels represented as strings to integers where there is one unique integer per class label (a common practice when applying machine learning).
- **Line 4:** We'll import the `train_test_split` function, which is a handy convenience function used to help us create our training and testing splits.
- **Line 5:** The `classification_report` function is another utility function that is used to help us evaluate the performance of our classifier and print a nicely formatted table of results to our console.

You can also see our implementations of the `SimplePreprocessor` and `SimpleDatasetLoader` imported on **Lines 6 and Line 7**, respectively.

Next, let's parse our command line arguments:

```
11
12  ap = argparse.ArgumentParser()
13  ap.add_argument(" d", "  dataset", required=    ,
14       help="path to input dataset")
15  ap.add_argument(" k", "  neighbors",     =    , default= ,
16       help="# of nearest neighbors for classification")
17  ap.add_argument(" j", "  jobs",     =    , default=- ,
18       help="# of jobs for k NN distance ( 1 uses all available cores)")
19  args =     (ap.parse_args())
```

Our script requires one command line argument, followed by two optional ones, each reviewed below:

- `--dataset`: The path to where our input image dataset resides on disk.
- `--neighbors`: Optional, the number of neighbors k to apply when using the k-NN algorithm.
- `--jobs`: Optional, the number of concurrent jobs to run when computing the distance between an input data point and the training set. A value of `-1` will use all available cores on the processor.

Now that our command line arguments are parsed, we can grab the file paths of the images in our dataset, followed by loading and preprocessing them (Step #1 in the classification pipeline):

```
21
22      ("[INFO] loading images...")
23  imagePaths =     (paths.list_images(args["dataset"]))
24
25
26
27  sp = SimplePreprocessor(  ,   )
28  sdl = SimpleDatasetLoader(preprocessors=[sp])
29  (data, labels) = sdl.load(imagePaths, verbose=   )
30  data = data.reshape((data.shape[ ],     ))
31
32
33      ("[INFO] features matrix: {:.1f}MB".format(
34      data.nbytes / (     *       )))
```

Line 23 grabs the file paths to all images in our dataset. We then initialize our `SimplePreprocessor` used to resize each image to 32×32 pixels on **Line 27**.

The `SimpleDatasetLoader` is initialized on **Line 28**, supplying our instantiated `SimplePreprocessor` as an argument (implying that `sp` will be applied to *every image* in the dataset).

A call to `.load` on **Line 29** loads our actual image dataset from disk. This method returns a 2-tuple of our `data` (each image resized to 32×32 pixels) along with the `labels` for each image.

After loading our images from disk, the `data` NumPy array has a `.shape` of `(3000, 32, 32, 3)`, indicating there are 3,000 images in the dataset, each 32×32 pixels with 3 channels.

However, in order to apply the k-NN algorithm, we need to "flatten" our images from a 3D representation to a single list of pixel intensities. We accomplish this, **Line 30** calls the `.reshape` method on the `data` NumPy array, flattening the $32 \times 32 \times 3$ images into an array with shape `(3000, 3072)`. The actual image data hasn't changed at all – the images are simply represented as a list of 3,000 entries, each of 3,072-dim ($32 \times 32 \times 3 = 3,072$).

To demonstrate how much memory it takes to store these 3,000 images in memory, **Lines 33 and 34** compute the number of bytes the array consumes and then converts the number to megabytes).

Next, let's building our training and testing splits (Step #2 in our pipeline):

```
36
37  le = LabelEncoder()
38  labels = le.fit_transform(labels)
39
40
41
42  (trainX, testX, trainY, testY) = train_test_split(data, labels,
43      test_size=    , random_state=  )
```

Lines 37 and 38 convert our `labels` (represented as strings) to integers where we have *one unique integer* per class. This conversion allows us to map the *cat* class to the integer 0, the *dog* class to integer 1, and the *panda* class to integer 2. Many machine learning algorithms assume that the class labels are encoded as integers, so it's important that we get in the habit of performing this step.

Computing our training and testing splits is handled by the `train_test_split` function on **Lines 42 and 43**. Here we partition our `data` and `labels` into two unique sets: 75% of the data for training and 25% for testing.

It is common to use the variable X to refer to a dataset that contains the data points we'll use for training and testing while y refers to the class labels (you'll learn more about this in Chapter 8 on *parameterized learning*). Therefore, we use the variables `trainX` and `testX` to refer to the *training and testing examples*, respectively. The variables `trainY` and `testY` are our *training and testing labels*. You will see these common notations throughout this book *and* in other machine learning books, courses, and tutorials that you may read.

Finally, we are able to create our k-NN classifier and evaluate it (Steps #3 and #4 in the image classification pipeline):

```
45
46      ("[INFO] evaluating k NN classifier...")
47  model = KNeighborsClassifier(n_neighbors=args["neighbors"],
48      n_jobs=args["jobs"])
49  model.fit(trainX, trainY)
50      (classification_report(testY, model.predict(testX),
51      target_names=le.classes_))
```

Lines 47 and 48 initialize the `KNeighborsClassifier` class. A call to the `.fit` method on **Line 49** "trains" the classifier, although there is no actual "learning" going on here – the k-NN model is simply storing the `trainX` and `trainY` data internally so it can create predictions on the testing set by computing the distance between the input data and the `trainX` data.

Lines 50 and 51 evaluate our classifier by using the `classification_report` function. Here we need to supply the `testY` class labels, the *predicted class labels* from our `model`, and optionally the names of the class labels (i.e., "dog", "cat", "panda").

k-NN Results

To run our k-NN classifier, execute the following command:

```
python knn.py --dataset ../datasets/animals
```

You should then see the following output similar to the following:

6.1 Libraries and Packages **63**
6.1.1 Python 63
6.1.2 Keras 64
6.1.3 Mxnet 64
6.1.4 OpenCV, scikit-image, scikit-learn, and more 64
6.2 Configuring Your Development Environment? **64**
6.3 Preconfigured Virtual Machine **65**
6.4 Cloud-based Instances **65**
6.5 How to Structure Your Projects **65**
6.6 Summary **66**

7.1 Working with Image Datasets **67**
7.1.1 Introducing the "Animals" Dataset 67
7.1.2 The Start to Our Deep Learning Toolkit 68
7.1.3 A Basic Image Preprocessor 69
7.1.4 Building an Image Loader 70
7.2 k-NN: A Simple Classifier **72**
7.2.1 A Worked k-NN Example 74
7.2.2 k-NN Hyperparameters 75
7.2.3 Implementing k-NN 75
7.2.4 k-NN Results 78
7.2.5 Pros and Cons of k-NN 79
7.3 Summary **80**

8.1 An Introduction to Linear Classification **82**
8.1.1 Four Components of Parameterized Learning 82
8.1.2 Linear Classification: From Images to Labels 83
8.1.3 Advantages of Parameterized Learning and Linear Classification 84
8.1.4 A Simple Linear Classifier With Python 85
8.2 The Role of Loss Functions **88**
8.2.1 What Are Loss Functions? 88
8.2.2 Multi-class SVM Loss 89
8.2.3 Cross-entropy Loss and Softmax Classifiers 91
8.3 Summary **94**

9.1 Gradient Descent **96**
9.1.1 The Loss Landscape and Optimization Surface 96
9.1.2 The "Gradient" in Gradient Descent 97
9.1.3 Treat It Like a Convex Problem (Even if It's Not) 98
9.1.4 The Bias Trick 98
9.1.5 Pseudocode for Gradient Descent 99
9.1.6 Implementing Basic Gradient Descent in Python 100

Notice how our feature matrix only consumes 9MB of memory for 3,000 images, each of size $32 \times 32 \times 3$ – this dataset can *easily* be stored in memory on modern machines without a problem.

Evaluating our classifier, we see that we obtained *52%* accuracy – this accuracy isn't bad for a classifier that doesn't do any true "learning" at all, given that the probability of randomly guessing the correct answer is $1/3$.

However, it is interesting to inspect the accuracy for each of the class labels. The "panda" class was correctly classified 79% of the time, likely due to the fact that pandas are largely black and white and thus these images lie closer together in our 3,072-dim space.

Dogs and cats obtain substantially lower classification accuracy at 39% and 36%, respectively. These results can be attributed to the fact that dogs and cats can have very similar shades of fur coats and the color of their coats cannot be used to discriminate between them. Background noise (such as grass in a backyard, the color of a couch an animal is resting on, etc.) can also "confuse" the k-NN algorithm as its unable to learn any discriminating patterns between these species. This confusion is one of the primary drawbacks of the k-NN algorithm: *while it's simple, it is also unable to learn from the data.*

Our next chapter will discuss the concept of *parameterized learning* where we can actually learn patterns from the *images themselves* rather than assuming images with similar contents will group together in an n-dimensional space.

Pros and Cons of k-NN

One main advantage of the k-NN algorithm is that it's *extremely simple* to implement and understand. Furthermore, the classifier takes absolutely no time to train, since all we need to do is store our data points for the purpose of later computing distances to them and obtaining our final classification.

However, we pay for this simplicity at classification time. Classifying a new testing point requires a comparison to *every single data point* in our training data, which scales $O(N)$, making working with larger datasets computationally prohibitive.

We can combat this time cost by using **Approximate Nearest Neighbor** (ANN) algorithms (such as kd-trees [65], FLANN [66], random projections [67, 68, 69], etc.); however, using these algorithms require that we trade space/time complexity for the "correctness" of our nearest neighbor algorithm since we are performing an approximation. That said, in many cases, it is well worth the effort and small loss in accuracy to use the k-NN algorithm. This behavior is in contrast to most machine learning algorithms (and *all* neural networks), where we spend a large amount of time upfront training our model to obtain high accuracy, and, in turn, have *very fast* classifications at testing time.

Finally, the k-NN algorithm is more suited for low-dimensional feature spaces (which images are not). Distances in high-dimensional feature spaces are often unintuitive, which you can read more about in Pedro Domingo's excellent paper [70].

It's also important to note that the k-NN algorithm doesn't actually "learn" anything – the algorithm is not able to make itself smarter if it makes mistakes: it's simply relying on distances in an n-dimensional space to make the classification.

Given these cons, why bother even studying the k-NN algorithm? The reason is that the algorithm is *simple*. It's *easy to understand*. And most importantly, it gives us a *baseline* that we

can use to compare neural networks and Convolutional Neural Networks to as we progress through the rest of this book.

Summary

In this chapter, we learned how to build a simple image processor and load an image dataset into memory. We then discussed the *k-Nearest Neighbor classifier*, or *k-NN* for short.

The k-NN algorithm classifies unknown data points by comparing the unknown data point to each data point in the training set. The comparison is done using a distance function or similarity metric. Then, from the most k similar examples in the training set, we accumulate the number of "votes" for each label. The category with the highest number of votes "wins" and is chosen as the overall classification.

While simple and intuitive, the k-NN algorithm has a number of drawbacks. The first is that it doesn't actually "learn" anything – if the algorithm makes a mistake, it has no way to "correct" and "improve" itself for later classifications. Secondly, without specialized data structures, the k-NN algorithm scales linearly with the number of data points, making it not only practically challenging to use in high dimensions, but theoretically questionable in terms of its usage [70].

Now that we have obtained a baseline for image classification using the k-NN algorithm, we can move on the *parameterized learning*, the foundation on which all deep learning and neural networks are built on. Using parameterized learning, we can actually *learn* from our input data and discover underlying patterns. This process will enable us to build high accuracy image classifiers that blow the performance of k-NN out of the water.

8. Parameterized Learning

In our previous chapter we learned about the k-NN classifier – a machine learning model so simple that it doesn't do any actual "learning" at all. We simply have to store the training data inside the model, and then predictions are made at test time by comparing the testing data points to our training data.

We've already discussed many of the pros and cons of k-NN, but in the context of large-scale datasets and deep learning, the most prohibitive aspect of k-NN is *the data itself*. While training may be simple, testing is quite slow, with the bottleneck being the distance computation between vectors. Computing the distances between training and testing points scales linearly with the number of points in our dataset, making the method impractical when our datasets become quite large. And while we can apply Approximate Nearest Neighbor methods such as ANN [71], FLANN [66], or Annoy [72], to speed up the search, that still doesn't alleviate the problem that k-NN cannot function without maintaining a replica of data inside the instantiation (or at least have a pointer to training set on disk, etc.)

To see why storing an exact replica of the training data inside the model is an issue, consider training a k-NN model and then deploying it to a customer base of 100, 1,000, or even 1,000,000 users. If your training set is only a few megabytes, this may not be a problem – but if your training set is measured in *gigabytes* to *terabytes* (as is the case for many datasets that we apply deep learning to), you have a real problem on your hands.

Consider the training set of the ImageNet dataset [42] which includes over 1.2 million images. If we trained a k-NN model on this dataset and then tried to deploy it to a set of users, we would need these users to download the k-NN model which internally represents *replicas* of the 1.2 million images. Depending on how you compress and store the data, this model could measure in hundreds of gigabytes to terabytes in storage costs and network overhead. Not only is this a waste of resources, its also not optimal for constructing a machine learning model.

Instead, a more desirable approach would be to 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 benefit of being defined by a *small number of parameters* that can easily be used to represent the model, *regardless of training size*. This type of machine learning is called

parameterized learning, which is defined as:

> *"A learning model that summarizes data with a set of parameters of fixed size (independent of the number of training examples) is called a parametric model. No matter how much data you throw at the parametric model, it won't change its mind about how many parameters it needs."* – Russell and Norvig (2009) [73]

In this chapter, we'll review the concept of parameterized learning and discuss how to implement a simple linear classifier. As we'll see later in this book, parameterized learning is the cornerstone of modern day machine learning and deep learning algorithms.

> Much of this chapter was inspired by Andrej Karpathy's excellent Linear Classification notes inside Stanford's cs231n class [74]. A big thank you to Karpathy and the rest of the cs231n teaching assistants for putting together such accessible notes.

An Introduction to Linear Classification

The first half of this chapter focuses on the fundamental theory and mathematics surrounding *linear classification* – and in general, *parameterized classification algorithms* that learn patterns from their training data. From there, I provide an actual linear classification implementation and example in Python so we can see how these types of algorithms work in code.

Four Components of Parameterized Learning

I've used the word "parameterized" a few times now, but what exactly does it mean?

Simply put: *parameterization* is the process of defining the necessary parameters of a given model. In the task of machine learning, parameterization involves defining a problem in terms of four key components: *data*, a *scoring function*, a *loss function*, and *weights and biases*. We'll review each of these below.

Data

This component is our *input data* that we are going to learn from. This data includes *both* the data points (i.e., raw pixel intensities from images, extracted features, etc.) and their associated class labels. Typically we denote our data in terms of a multi-dimensional **design matrix** [10].

Each row in the design matrix represents a data point while each column (which itself could be a multi-dimensional array) of the matrix corresponds to a different feature. For example, consider a dataset of 100 images in the RGB color space, each image sized 32×32 pixels. The design matrix for this dataset would be $X \subseteq R^{100 \cdot (32 \cdot 32 \cdot 3)}$ where X_i defines the i-th image in R. Using this notation, X_1 is the first image, X_2 the second image, and so on.

Along with the design matrix, we also define a vector y where y_i provides the class label for the i-th example in the dataset.

Scoring Function

The scoring function accepts our data as an input and maps the data to class labels. For instance, given our set of input images, the scoring function takes these data points, applies some function f (our scoring function), and then returns the predicted class labels, similar to the pseudocode below:

```
INPUT_IMAGES => F(INPUT_IMAGES) => OUTPUT_CLASS_LABELS
```

Loss Function

A loss function quantifies how well our *predicted class labels* agree with our *ground-truth labels*. The higher level of agreement between these two sets of labels, the *lower our loss* (and higher our classification accuracy, at least on the training set).

Our goal when training a machine learning model is to *minimize the loss function*, thereby increasing our classification accuracy.

Weights and Biases

The weight matrix, typically denoted as $\boldsymbol{W}$ and the bias vector $\boldsymbol{b}$ are called the **weights** or **parameters** of our classifier that we'll actually be optimizing. Based on the output of our scoring function and loss function, we'll be tweaking and fiddling with the values of the weights and biases to increase classification accuracy.

Depending on your model type, there may exist many more parameters, but at the most basic level, these are the four building blocks of parameterized learning that you'll commonly encounter. Once we've defined these four key components, we can then apply optimization methods that allow us to find a set of parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ that minimize our loss function with respect to our scoring function (while increasing classification accuracy on our data).

Next, let's look at how these components can work together to build a linear classifier, transforming the input data into actual predictions.

Linear Classification: From Images to Labels

In this section, we are going to look at a more mathematical motivation of the parameterized model approach to machine learning.

To start, we need our ***data***. Let's assume that our training dataset is denoted as x_i where each image has an associated class label y_i. We'll assume that $i = 1, \ldots, N$ and $y_i = 1, \ldots, K$, implying that we have N data points of dimensionality D, separated into K unique categories.

To make this idea more concrete, consider our "Animals" dataset from Chapter 7. In this dataset, we have $N = 3{,}000$ total images. Each image is 32×32 pixels, represented in the RGB color space (i.e., three channels per image). We can represent each image as $D = 32 \times 32 \times 3 = 3{,}072$ distinct values. Finally, we know there are a total of $K = 3$ class labels: one for the dog, cat, and panda classes, respectively.

Given these variables, we must now define a scoring function f that maps the images to the class label scores. One method to accomplish this scoring is via a simple linear mapping:

$$f(x_i, W, b) = Wx_i + b \tag{8.1}$$

Let's assume that each x_i is represented as a single column vector with shape $[D \times 1]$ (in this example we would flatten the $32 \times 32 \times 3$ image into a list of 3,072 integers). Our weight matrix $\boldsymbol{W}$ would then have a shape of $[K \times D]$ (the number of class labels by the dimensionality of the input images). Finally $\boldsymbol{b}$, the **bias vector** would be of size $[K \times 1]$. The bias vector allows us to shift and translate our scoring function in one direction or another without actually influencing our weight matrix $\boldsymbol{W}$. The bias parameter is often critical for successful learning.

Going back to the Animals dataset example, each x_i is represented by a list of 3,072 pixel values, so x_i, therefore, has the shape $[3{,}072 \times 1]$. The weight matrix $\boldsymbol{W}$ will have a shape of $[3 \times 3{,}072]$ and finally the bias vector $\boldsymbol{b}$ will be of size $[3 \times 1]$.

Figure 8.1 follows an illustration of the linear classification scoring function f. On the *left*, we have our original input image, represented as a $32 \times 32 \times 3$ image. We then *flatten* this image into a list of 3,072 pixel intensities by taking the 3D array and reshaping it into a 1D list.

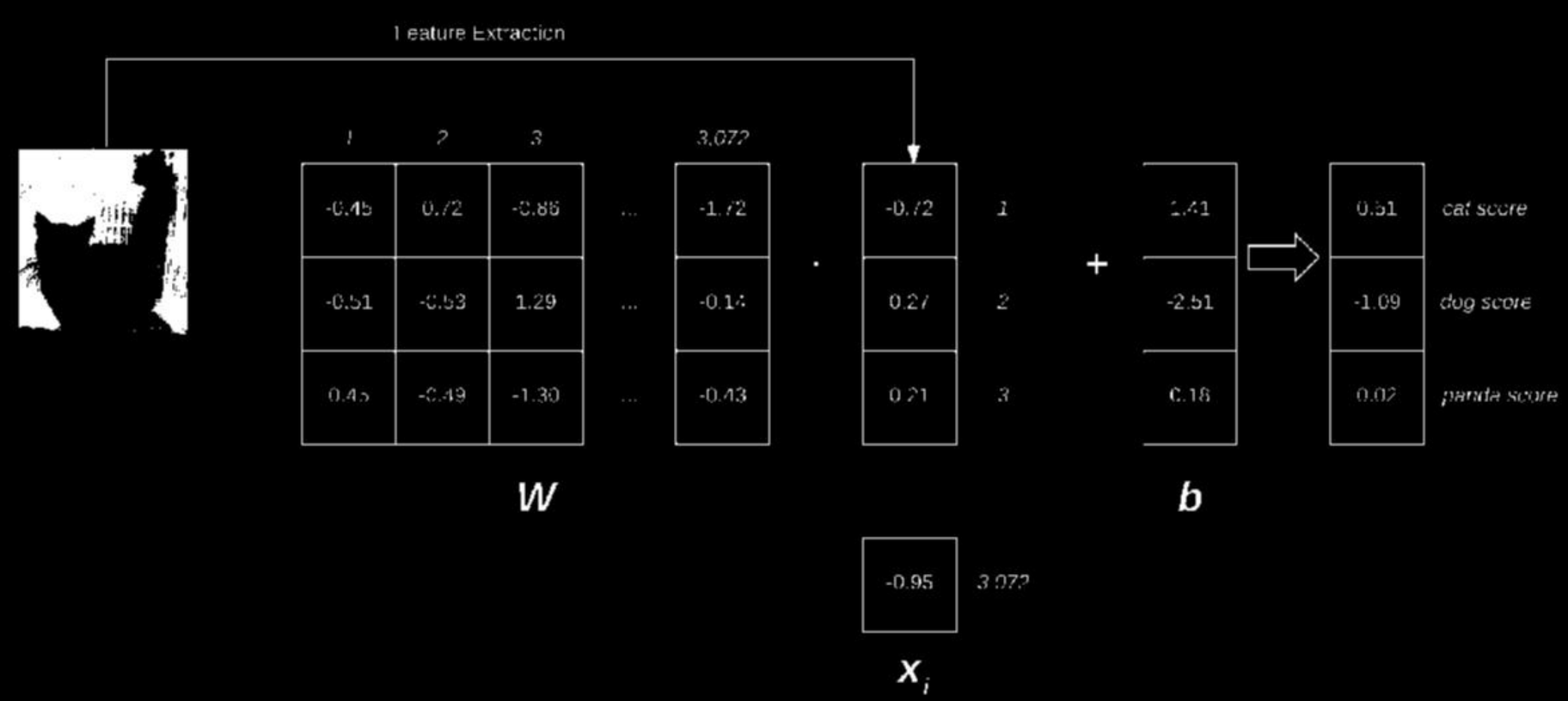

Figure 8.1: Illustrating the dot product of the weight matrix $\boldsymbol{W}$ and feature vector $\boldsymbol{x}$, followed by the addition of the bias term. Figure inspired by Karpathy's example in Stanford University's cs231n course [57].

Our weight matrix $\boldsymbol{W}$ contains three rows (one for each class label) and 3,072 columns (one for each of the pixels in the image). After taking the dot product between $\boldsymbol{W}$ and x_i, we add in the bias vector $\boldsymbol{b}$ – the result is our actual **scoring function**. Our scoring function yields three values on the *right*: the scores associated with the dog, cat, and panda labels, respectively.

> Readers who are unfamiliar with taking dot products should read this quick and concise tutorial: . For readers interested in studying linear algebra in depth, I highly recommend working through *Coding the Matrix Linear Algebra through Applications to Computer Science* by Philip N. Klein [75].

Looking at the above figure and equation, you can convince yourself that the input x_i and y_i are *fixed* and *not something we can modify*. Sure, we can obtain different x_is by applying various transformations to the input image – but once we pass the image into the scoring function, **these values do not change**. In fact, the only parameters that we have any control over (in terms of parameterized learning) are our weight matrix $\boldsymbol{W}$ and our bias vector $\boldsymbol{b}$. Therefore, our goal is to utilize both our scoring function and loss function to *optimize* (i.e., modify in a systematic way) the weight and bias vectors such that our classification accuracy *increases*.

Exactly *how* we optimize the weight matrix depends on our loss function, but typically involves some form of gradient descent. We'll be reviewing loss functions later in this chapter. Optimization methods such as gradient descent (and its variants) will be discussed in Chapter 9. However, for the time being, simply understand that given a scoring function, we will also define a loss function that tells us how "good" our predictions are on the input data.

Advantages of Parameterized Learning and Linear Classification

There are two primary advantages to utilizing parameterized learning:

1. **Once we are done training our model, we can discard the input data and keep only the weight matrix W and the bias vector b.** This *substantially* reduces the size of our model since we need to store two sets of vectors (versus the *entire* training set).
2. **Classifying new test data is *fast*.** In order to perform a classification, all we need to do is take the dot product of $\boldsymbol{W}$ and x_i, follow by adding in the bias $\boldsymbol{b}$ (i.e., apply our scoring

function). Doing it this way is *significantly faster* than needing to compare each testing point to *every* training example, as in the k-NN algorithm.

A Simple Linear Classifier With Python

Now that we've reviewed the concept of parameterized learning and linear classification, let's implement a *very simple* linear classifier using Python.

The purpose of this example is *not* to demonstrate how we train a model from start to finish (we'll be covering that in a later chapter as we still have some ground to cover before we're ready to train a model from scratch), but to simply show how we would initialize a weight matrix $\boldsymbol{W}$, bias vector $\boldsymbol{b}$, and then use these parameters to classify an image via a simple dot product.

Let's go ahead and get this example started. Our goal here is to write a Python script that will correctly classify Figure 8.2 as *"dog"*.

Figure 8.2: Our example input image that we are going to classifier with a simple linear classifier.

To see how we can accomplish this classification, open a new file, name it `linear_example.py`, and insert the following code:

```
1
2         numpy     np
3         cv2
4
5
6
7  labels = ["dog", "cat", "panda"]
8  np.random.seed( )
```

Lines 2 and 3 import our required Python packages. We'll use NumPy for our numerical processing and OpenCV to load our example image from disk.

Line 7 initializes the list of target class labels for the "Animals" dataset while **Line 8** sets the pseudorandom number generator for NumPy, ensuring that we can reproduce the results of this experiment.

Next, let's initialize our weight matrix and bias vector:

```
10
11
```

```
12
13
14  W = np.random.randn( ,     )
15  b = np.random.randn( )
```

Line 14 initializes the weight matrix `W` with random values from a uniform distribution, sampled over the range $[0, 1]$. This weight matrix has 3 rows (one for each of the class labels) and 3072 columns (one for each of the pixels in our $32 \times 32 \times 3$ image).

We then initialize the bias vector on **Line 15** – this vector is also randomly filled with values uniformly sampled over the distribution $[0, 1]$. Our bias vector has 3 rows (corresponding to the number of class labels) along with one column.

If we were training this linear classifier *from scratch* we would need to *learn* the values of `W` and `b` through an optimization process. However, since we have not reached the optimization stage of training a model, I have initialized the pseudorandom number generator with a value `1` to ensure the random values give us the "correct" classification (I tested random initialization values ahead of time to determine which value gives us the correct classification). For the time being, simply treat the weight matrix `W` and the bias vector `b` as "black box arrays" that are optimized in a magical way – we'll pull back the curtain and reveal how these parameters are learned in the next chapter.

Now that our weight matrix and bias vector are initialized, let's load our example image from disk:

```
17
18
19  orig = cv2.imread("beagle.png")
20  image = cv2.resize(orig, (   ,   )).flatten()
```

Line 19 loads our image from disk via `cv2.imread`. We then resize the image to 32×32 pixels (ignoring the aspect ratio) on **Line 20** – our image is now represented as a `(32, 32, 3)` NumPy array, which we flatten into 3,072-dim vector.

The next step is to compute the output class label scores by applying our scoring function:

```
22
23
24  scores = W.dot(image) + b
```

Line 24 is the scoring function itself – it's simply the dot product between the weight matrix `W` and the input image pixel intensities, followed by adding in the bias `b`.

Finally, our last code block handles writing the scoring function values for each of the class labels to our terminal, then displaying the result to our screen:

```
26
27      (label, score)       (labels, scores):
28            ("[INFO] {}: {:.2f}".format(label, score))
29
30
31
32  cv2.putText(orig, "Label: {}".format(labels[np.argmax(scores)]),
33       (   ,   ), cv2.FONT_HERSHEY_SIMPLEX,    , ( ,    ,  ),  )
34
```

```
35
36 cv2.imshow("Image", orig)
37 cv2.waitKey( )
```

To execute our example, just issue the following command:

```
python linear_example.py
```

Notice how the *dog* class has the *largest scoring function value*, which implies that the "dog" class would be chosen as the prediction by our classifier. In fact, we can see the text dog correctly drawn on our input image (Figure 8.2) in Figure 8.3.

Figure 8.3: In this example, our linear classifier was correctly able to label the input image as *dog*; however, keep in mind that this is a worked example. Later in this book you'll learn how to *train* our weights and biases to automatically make these predictions.

Again, keep in mind that this was a *worked example*. I ***purposely*** set the random state of our Python script to generate `W` and `b` values that would lead to the correct classification (you can change the pseudorandom seed value on **Line 8** to see for yourself how different random initializations will produce different output predictions).

In practice, you would *never* initialize your `W` and `b` values and *assume* they would give you the correct classification without some sort of learning process. Instead, when training our own machine learning models from scratch we would need to *optimize* and *learn* `W` and `b` via an optimization algorithm, such as gradient descent.

We'll cover optimization and gradient descent in the next chapter, but in the meantime, simply take the time to ensure you understand **Line 24** and how a linear classifier makes a classification by taking the dot product between a weight matrix and an input data point, followed by adding in the bias. Our *entire model* can therefore be defined via two values: the weight matrix and the bias vector. This representation is not only *compact* but also quite *powerful* when we train machine learning models from scratch.

The Role of Loss Functions

In our last section we discussed the concept of parameterized learning. This type of learning allows us to take sets of input data and class labels, and actually learn a function that *maps* the input to the output predictions by defining a set of parameters and optimizing over them.

But in order to actually "learn" the mapping from the input data to class labels via our scoring function, we need to discuss two important concepts:

1. Loss functions
2. Optimization methods

The rest of this chapter is dedicated to common loss functions you'll encounter when building neural networks and deep learning networks. Chapter 9 of the *Starter Bundle* is dedicated entirely to basic optimization methods while Chapter 7 of the *Practitioner Bundle* discusses more advanced optimization methods.

Again, this chapter is meant to be a *brief review* of loss functions and their role in parameterized learning. A thorough discussion of loss functions is outside the scope of this book and I would highly recommend Andrew Ng's Coursera course [76], Witten et al. [77], Harrington [78], and Marsland [79] if you would like to complement this chapter with more mathematically rigorous derivations.

What Are Loss Functions?

Figure 8.4: The training losses for two separate models trained on the CIFAR-10 dataset are plotted over time. Our loss function quantifies how "good" or "bad" of a job a given model is doing at classifying data points from the dataset. Model #1 achieves considerably lower loss than Model #2.

At the most basic level, a loss function quantifies how "good" or "bad" a given predictor is at classifying the input data points in a dataset. A visualization of loss functions plotted over time

9.1.7 Simple Gradient Descent Results 104
9.2 Stochastic Gradient Descent (SGD) 106
9.2.1 Mini-batch SGD 106
9.2.2 Implementing Mini-batch SGD 107
9.2.3 SGD Results 110
9.3 Extensions to SGD 111
9.3.1 Momentum 111
9.3.2 Nesterov's Acceleration 112
9.3.3 Anecdotal Recommendations 113
9.4 Regularization 113
9.4.1 What Is Regularization and Why Do We Need It? 113
9.4.2 Updating Our Loss and Weight Update To Include Regularization 115
9.4.3 Types of Regularization Techniques 116
9.4.4 Regularization Applied to Image Classification 117
9.5 Summary 119

10.1 Neural Network Basics 121
10.1.1 Introduction to Neural Networks 122
10.1.2 The Perceptron Algorithm 129
10.1.3 Backpropagation and Multi-layer Networks 137
10.1.4 Multi-layer Networks with Keras 153
10.1.5 The Four Ingredients in a Neural Network Recipe 163
10.1.6 Weight Initialization 165
10.1.7 Constant Initialization 165
10.1.8 Uniform and Normal Distributions 165
10.1.9 LeCun Uniform and Normal 166
10.1.10 Glorot/Xavier Uniform and Normal 166
10.1.11 He et al./Kaiming/MSRA Uniform and Normal 167
10.1.12 Differences in Initialization Implementation 167
10.2 Summary 168

11.1 Understanding Convolutions 170
11.1.1 Convolutions versus Cross-correlation 170
11.1.2 The "Big Matrix" and "Tiny Matrix' Analogy 171
11.1.3 Kernels 171
11.1.4 A Hand Computation Example of Convolution 172
11.1.5 Implementing Convolutions with Python 173
11.1.6 The Role of Convolutions in Deep Learning 179
11.2 CNN Building Blocks 179
11.2.1 Layer Types 181
11.2.2 Convolutional Layers 181
11.2.3 Activation Layers 186
11.2.4 Pooling Layers 186
11.2.5 Fully-connected Layers 188
11.2.6 Batch Normalization 189
11.2.7 Dropout 190

for two separate models trained on the CIFAR-10 dataset is shown in Figure 8.4. The smaller the loss, the better a job the classifier is at modeling the relationship between the input data and output class labels (although there is a point where we can *overfit* our model – by modeling the training data *too closely*, our model loses the ability to generalize, a phenomenon we'll discuss in detail in Chapter 17). Conversely, the larger our loss, the *more work needs to be done* to increase classification accuracy.

To improve our classification accuracy, we need to tune the parameters of our weight matrix $\boldsymbol{W}$ or bias vector $\boldsymbol{b}$. Exactly *how* we go about updating these parameters is an *optimization problem*, which we'll be covering in the next chapter. For the time being, simply understand that a *loss function* can be used to quantify how well our *scoring function* is doing at classifying input data points.

Ideally, our loss should decrease over time as we tune our model parameters. As Figure 8.4 demonstrates, Model #1's loss starts slightly higher than Model #2, but then decreases rapidly and continues to stay low when trained on the CIFAR-10 dataset. Conversely, the loss for Model #2 decreases initially but quickly stagnates. In this specific example, Model #1 is achieving lower overall loss and is likely a more desirable model to be used on classifying other images from the CIFAR-10 dataset. I say "likely" because there is a chance that Model #1 has overfit to the training data. We'll cover this concept of overfitting and how to spot it in Chapter 17.

Multi-class SVM Loss

Multi-class SVM Loss (as the name suggests) is inspired by (Linear) Support Vector Machines (SVMs) [43] which uses a scoring function f to map our data points to numerical scores for each class labels. This function f is a simple learning mapping:

$$f(x_i, W, b) = Wx_i + b \tag{8.2}$$

Now that we have our scoring function, we need to determine how "good" or "bad" this function is (given the weight matrix $\boldsymbol{W}$ and bias vector $\boldsymbol{b}$) at making predictions. To make this determination, we need a *loss function*.

Recall that when creating a machine learning model we have a *design matrix* $\boldsymbol{X}$, where each row in $\boldsymbol{X}$ contains a data point we wish to classify. In the context of image classification, each row in $\boldsymbol{X}$ is an image and we seek to correctly label this image. We can access the i-th image inside $\boldsymbol{X}$ via the syntax x_i.

Similarly, we also have a vector $\boldsymbol{y}$ which contains our class labels for each $\boldsymbol{X}$. These $\boldsymbol{y}$ values are our *ground-truth labels* and what we hope our scoring function will correctly predict. Just like we can access a given image as x_i, we can access the associated class label via y_i.

As a matter of simplicity, let's abbreviate our scoring function as s:

$$s = f(x_i, W) \tag{8.3}$$

Which implies that we can obtain the predicted score of the j-th class via the i-th data point:

$$s_j = f(x_i, W)_j \tag{8.4}$$

Using this syntax, we can put it all together, obtaining the ***hinge loss function***:

$$L_i = \sum_{j \neq y_i} max(0, s_j - s_{y_i} - 1) \tag{8.5}$$

Nearly all loss functions include a regularization term. I am skipping this idea for now as we'll review regularization in Chapter 9 once we better understand loss functions.

Looking at the hinge loss equation above, you might be confused at what it's actually doing. Essentially, the hinge loss function is summing across all *incorrect classes* ($i \neq j$) and comparing the output of our scoring function s returned for the j-th class label (the incorrect class) and the y_i-th class (the correct class). We apply the *max* operation to clamp values at zero, which is important to ensure we do not sum negative values.

A given x_i is classified correctly when the loss $L_i = 0$ (I'll provide a numerical example in the following section). To derive the loss across our *entire training set*, we simply take the mean over each individual L_i:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i \tag{8.6}$$

Another related loss function you may encounter is the ***squared hinge loss***:

$$L_i = \sum_{j \neq y_i} max(0, s_j - s_{y_i} - 1)^2 \tag{8.7}$$

The squared term penalizes our loss more heavily by squaring the output, which leads to quadratic growth in loss in a prediction that is incorrect (versus a linear growth).

As for which loss function you should use, that is entirely dependent on your dataset. It's typical to see the standard hinge loss function used more, but on some datasets the squared variation may obtain better accuracy. Overall, this is a *hyperparameter* that you should consider tuning.

A Multi-class SVM Loss Example

Now that we've taken a look at the mathematics behind hinge loss, let's examine a worked example. We'll again be using the "Animals" dataset which aims to classify a given image as containing a *cat*, *dog*, or *panda*. To start, take a look at Figure 8.5 where I have included three training examples from the three classes of the "Animals" dataset.

Given some arbitrary weight matrix $\boldsymbol{W}$ and bias vector $\boldsymbol{b}$, the output scores of $f(x, W) = Wx - b$ are displayed in the body of the matrix. The *larger* the scores are, the *more confident* our scoring function is regarding the prediction.

Let's start by computing the loss L_i for the "dog" class:

```
>>>    ( ,      -      + ) +    ( , -     -      + )
```

Notice how our equation here includes two terms – the difference between the predicted dog score and *both* the cat and panda score. Also observe how the loss for "dog" is *zero* – this implies that the dog was correctly predicted. A quick investigation of Image #1 from Figure 8.5 above demonstrates this result to be true: the "dog" score is greater than both the "cat" and "panda" scores.

Similarly, we can compute the hinge loss for Image #2, this one containing a cat:

```
>>>    ( ,      - (-    ) +  ) +    ( , -     - (-    ) +  )
```

	Image #1	Image #2	Image #3
Dog	4.26	3.76	-2.37
Cat	1.33	-1.20	1.03
Panda	-1.01	-3.81	-2.27

Figure 8.5: At the top of the figure we have three input images: one for each of the dog, cat, and panda class, respectively. The body of the table contains the scoring function outputs for each of the classes. We will use the scoring function to derive the total loss for each input image.

In this case, our loss function is greater than zero, indicating that our prediction is *incorrect*. Looking at our scoring function, we see that our model predicts *dog* as the proposed label with a score of 3.76 (as this is the label with the highest score). We know that this label is incorrect – and in Chapter 9 we'll learn how to automatically tune our weights to correct these predictions.

Finally, let's compute the hinge loss for the panda example:

```
>>>    ( , -      - (-     ) +  ) +    ( ,       - (-     ) +  )
```

Again, our loss is non-zero, so we know we have an incorrect prediction. Looking at our scoring function, our model has incorrectly labeled this image as "cat" when it should be "panda".

We can then obtain the *total loss* over the three examples by taking the average:

```
>>> (     +       +     ) /
```

Therefore, given our three training examples our overall hinge loss is 3.72 for the parameters $\boldsymbol{W}$ and $\boldsymbol{b}$.

Also take note that our loss was zero for only *one* of the three input images, implying that *two* of our predictions were incorrect. In our next chapter we'll learn how to optimize $\boldsymbol{W}$ and $\boldsymbol{b}$ to make better predictions by using the loss function to help drive and steer us in the right direction.

Cross-entropy Loss and Softmax Classifiers

While hinge loss is quite popular, you're *much* more likely to run into cross-entropy loss and Softmax classifiers in the context of deep learning and convolutional neural networks.

Why is this? Simply put:

Softmax classifiers give you *probabilities* for each class label while hinge loss gives you the *margin*.

It's much easier for us as humans to interpret *probabilities* rather than margin scores. Furthermore, for datasets such as ImageNet, we often look at the rank-5 accuracy of Convolutional Neural Networks (where we check to see if the ground-truth label is in the top-5 predicted labels returned by a network for a given input image). Seeing if (1) the true class label exists in the top-5 predictions and (2) the *probability* associated with each label is a nice property.

Understanding Cross-entropy Loss

The Softmax classifier is a generalization of the binary form of Logistic Regression. Just like in hinge loss or squared hinge loss, our mapping function f is defined such that it takes an input set of data x_i and maps them to output class labels via dot product of the data x_i and weight matrix $\boldsymbol{W}$ (omitting the bias term for brevity):

$$f(x_i, W) = Wx_i \tag{8.8}$$

However, unlike hinge loss, we can interpret these scores as *unnormalized log probabilities* for each class label, which amounts to swapping out the hinge loss function with cross-entropy loss:

$$L_i = -log(e^{s_{y_i}} / \sum_j e^{s_j}) \tag{8.9}$$

So, how did I arrive here? Let's break the function apart and take a look. To start, our loss function should minimize the negative log likelihood of the correct class:

$$L_i = -logP(Y = y_i | X = x_i) \tag{8.10}$$

The probability statement can be interpreted as:

$$P(Y = k | X = x_i) = e^{s_{y_i}} / \sum_j e^{s_j} \tag{8.11}$$

Where we use our standard scoring function form:

$$s = f(x_i, W) \tag{8.12}$$

As a whole, this yields our final loss function for a *single* data point, just like above:

$$L_i = -log(e^{s_{y_i}} / \sum_j e^{s_j}) \tag{8.13}$$

Take note that your logarithm here is actually base e (natural logarithm) since we are taking the inverse of the exponentiation over e earlier. The actual exponentiation and normalization via the sum of exponents is our *Softmax function*. The negative log yields our actual *cross-entropy loss*.

Just as in hinge loss and squared hinge loss, computing the cross-entropy loss over an entire dataset is done by taking the average:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i \tag{8.14}$$

Again, I'm purposely omitting the regularization term from our loss function. We'll return to regularization, explain what it is, how to use it, and why it's critical to neural networks and deep learning in Chapter 9. If the equations above seem scary, don't worry – we're about to work through numerical examples in the next section to ensure you understand how cross-entropy loss works.

A Worked Softmax Example

Input Image

	Scoring Function
Dog	-3.44
Cat	1.16
Panda	3.91

	Scoring Function	Unnormalized Probabilities
Dog	-3.44	0.03
Cat	1.16	3.19
Panda	3.91	49.90

	Scoring Function	Unnormalized Probabilities	Normalized Probabilities
Dog	-3.44	0.0321	0.0006
Cat	1.16	3.1899	0.0601
Panda	3.91	49.8990	0.9393

	Scoring Function	Unnormalized Probabilities	Normalized Probabilities	Negative Log Loss
Dog	-3.44	0.0321	0.0006	
Cat	1.16	3.1899	0.0601	
Panda	3.91	49.8990	0.9393	**0.0626**

Figure 8.6: **First Table:** To compute our cross-entropy loss, let's start with the output of the scoring function. **Second Table:** Exponentiating the output values from the scoring function gives us our unnormalized probabilities. **Third Table:** To obtain the actual probabilities, we divide each individual unnormalized probabilities by the sum of all unnormalized probabilities. **Fourth Table:** Taking the negative natural logarithm of the probability for the correct ground-truth yields the final loss for the data point.

To demonstrate cross-entropy loss in action, consider Figure 8.6. Our goal is to classify whether the image above contains a *dog*, *cat*, or *panda*. Clearly, we can see that the image is a "panda" – *but what does our Softmax classifier think?* To find out, we'll need to work through each of the four tables in the figure.

The **first table** includes the output of our scoring function f for each of the three classes, respectively. These values are our *unnormalized log probabilities* for the three classes. Let's exponentiate the output of the scoring function (e^s, where s is our score function value), yielding our *unnormalized probabilities* (**second table**).

The next step is to take the denominator, sum the exponents, and divide by the sum, thereby yielding the *actual probabilities associated with each class label* (**third table**). Notice how the probabilities sum to one. Finally, we can take the negative natural logarithm, $-ln(p)$, where p is the normalized probability, yielding our final loss (the **fourth and final table**).

In this case, our Softmax classifier would correctly report the image as *panda* with 93.93% confidence. We can then repeat this process for all images in our training set, take the average, and obtain the overall cross-entropy loss for the training set. This process allows us to quantify how good or bad a set of parameters are performing on our training set.

> I used a random number generator to obtain the score function values for this particular example. These values are simply used to demonstrate how the calculations of the Softmax classifier/cross-entropy loss function are performed. In reality, these values would *not* be randomly generated – they would instead be the output of your scoring function f based on your parameters $\boldsymbol{W}$ *and* $\boldsymbol{b}$. We'll see how all the components of parameterized learning fit together in our next chapter, but for the time being, we are working with example numbers to demonstrate how loss functions work.

Summary

In this chapter, we reviewed four components of parameterized learning:

1. Data
2. Scoring function
3. Loss function
4. Weights and biases

In the context of image classification, our input data is our dataset of images. The scoring function produces *predictions* for a given input image. The loss function then *quantifies* how good or bad a set of predictions are over the dataset. Finally, the weight matrix and bias vectors are what enable us to actually "learn" from the input data – these parameters will be tweaked and tuned via optimization methods in an attempt to obtain higher classification accuracy.

We then reviewed two popular loss functions: *hinge loss* and *cross-entropy loss*. While hinge loss is used in many machine learning applications (such as SVMs), I can almost guarantee with absolutely certainty that you'll see cross-entropy loss with more frequency primarily due to the fact that Softmax classifiers output *probabilities* rather than *margins*. Probabilities are much easier for us as humans to interpret, so this fact is a particularly nice quality of cross-entropy loss and Softmax classifiers. For more information on loss hinge loss and cross-entropy loss, please refer to Stanford University's cs231n course [57, 74].

In our next chapter we'll review optimization methods that are used to tune our weight matrix and bias vector. Optimization methods allow our algorithms to actually *learn* from our input data by updating the weight matrix and bias vector based on the output of our scoring and loss functions. Using these techniques we can take *incremental steps* towards parameter values that obtain lower loss and higher accuracy. Optimization methods are the cornerstone of modern day neural networks and deep learning, and without them, we would be unable to learn patterns from our input data, so be sure to pay attention to the upcoming chapter.

9. Optimization Methods and Regularization

"Nearly all of deep learning is powered by one very important algorithm: Stochastic Gradient Descent (SGD)" – Goodfellow et al. [10]

At this point we have a strong understanding of the concept of parameterized learning. Over the past few chapters, we have discussed the concept of *parameterized learning* and how this type of learning enables us to define a *scoring function* that maps our input data to output class labels.

This scoring function is defined in terms of two important *parameters*; specifically, our weight matrix $\boldsymbol{W}$ and our bias vector $\boldsymbol{b}$. Our scoring function accepts these parameters as inputs and returns a prediction for each input data point x_i.

We have also discussed two common loss functions: Multi-class SVM loss and cross-entropy loss. Loss functions, at the most basic level, are used to quantify how "good" or "bad" a given predictor (i.e., a set of parameters) is at classifying the input data points in our data.

Given these building blocks, we can now move on to the *most important aspect* of machine learning, neural networks, and deep learning – ***optimization***. Optimization algorithms are the engines that power neural networks and enable them to learn patterns from data. Throughout this discussion, we've learned that obtaining a high accuracy classifier is *dependent* on finding a set of weights $\boldsymbol{W}$ and $\boldsymbol{b}$ such that our data points are correctly classified.

But how do we go about *finding* and *obtaining* a weight matrix $\boldsymbol{W}$ and bias vector $\boldsymbol{b}$ that obtains high classification accuracy? Do we randomly initialize them, evaluate, and repeat over and over again, *hoping* that at *some point* we land on a set of parameters that obtains reasonable classification? We could – but given that modern deep learning networks have parameters that number in the tens of millions, it may take us a long time to blindly stumble upon a reasonable set of parameters.

Instead of relying on pure randomness, we need to define an *optimization algorithm* that allows us to *literally improve* $\boldsymbol{W}$ **and** $\boldsymbol{b}$. In this chapter, we'll be looking at the most common algorithm used to train neural networks and deep learning models – *gradient descent*. Gradient descent has many variants (which we'll also touch on), but, in each case, the idea is the same: iteratively evaluate your parameters, compute your loss, then take a small step in the direction that will minimize your loss.

Gradient Descent

The gradient descent algorithm has two primary flavors:

1. The standard "vanilla" implementation.
2. The optimized "stochastic" version that is more commonly used.

In this section we'll be reviewing the basic vanilla implementation to form a baseline for our understanding. After we understand the basics of gradient descent, we'll move on to the stochastic version. We'll then review some of the "bells and whistles" that we can add on to gradient descent, including momentum, and Nesterov acceleration.

The Loss Landscape and Optimization Surface

The gradient descent method is an *iterative optimization algorithm* that operates over a **loss landscape** (also called an *optimization surface*). The canonical gradient descent example is to visualize our weights along the x-axis and then the loss for a given set of weights along the y-axis (Figure 9.1, *left*):

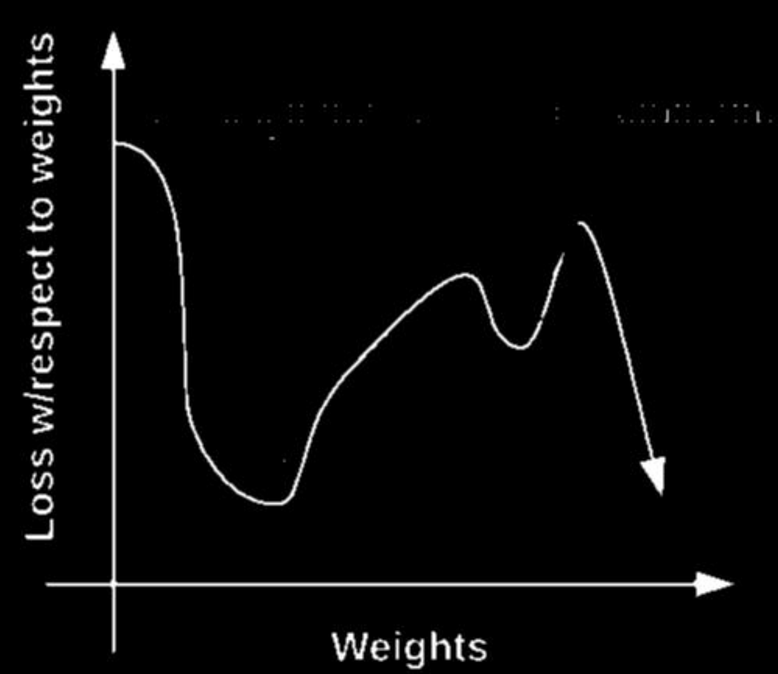

Figure 9.1: **Left:** The "naive loss" visualized as a 2D plot. **Right:** A more realistic loss landscape can be visualized as a bowl that exists in multiple dimensions. Our goal is to apply gradient descent to navigate to the bottom of this bowl (where there is low loss).

As we can see, our loss landscape has many peaks and valleys based on which values our parameters take on. Each peak is a *local maximum* that represents very high regions of loss – the local maximum with the largest loss across the entire loss landscape is the *global maximum*. Similarly, we also have *local minimum* which represents many small regions of loss.

The local minimum with the smallest loss across the loss landscape is our *global minimum*. In an ideal world, we would like to find this global minimum, ensuring our parameters take on the most optimal possible values.

So that raises the question: *"If we want to reach a global minimum, why not just directly jump to it? It's clearly visible on the plot?"*

Therein lies the problem – the loss landscape is invisible to us. We don't *actually* know what it looks like. If we're an optimization algorithm, we would be *blindly* placed somewhere on the plot, having no idea what the landscape in front of us looks like, and we would have to navigate our way to a loss minimum without accidentally climbing to the top of a local maximum.

Personally, I've never liked this visualization of the loss landscape – it's too simple, and it often leads readers to think that gradient descent (and its variants) will eventually find either a local or global minimum. *This statement isn't true, especially for complex problems* – and I'll explain why

later in this chapter. Instead, let's look at a different visualization of the loss landscape that I believe does a better job depicting the problem. Here we have a bowl, similar to the one you may eat cereal or soup out of (Figure 9.1, *right*).

The surface of our bowl is the loss landscape, which is a *plot* of the loss function. The difference between our loss landscape and your cereal bowl is that your cereal bowl only exists in three dimensions, while your loss landscape exists in *many dimension*, perhaps tens, hundreds, or thousands of dimensions.

Each position along the surface of the bowl corresponds to a *particular loss value* given a set of parameters $\boldsymbol{W}$ (weight matrix) and $\boldsymbol{b}$ (bias vector). Our goal is to try different values of $\boldsymbol{W}$ and $\boldsymbol{b}$, evaluate their loss, and then take a step towards more optimal values that (ideally) have lower loss.

The "Gradient" in Gradient Descent

To make our explanation of gradient descent a little more intuitive, let's pretend that we have a robot – let's name him Chad (Figure 9.2, *left*). When performing gradient descent, we randomly drop Chad somewhere on our loss landscape (Figure 9.2, *right*).

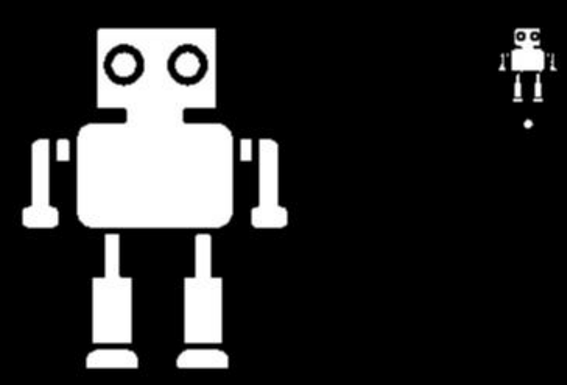

Figure 9.2: **Left:** Our robot, Chad. **Right:** It's Chad's job to navigate our loss landscape and descend to the bottom of the basin. Unfortunately, the only sensor Chad can use to control his navigation is a special function, called a *loss function*, L. This function must guide him to an area of lower loss.

It's now Chad's job to navigate to the bottom of the basin (where there is minimum loss). Seems easy enough right? All Chad has to do is orient himself such that he's facing "downhill" and ride the slope until he reaches the bottom of the bowl.

But here's the problem: Chad isn't a very smart robot. Chad has only one sensor – this sensor allows him to take his parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ and then compute a loss function L. Therefore, Chad is able to compute his relative position on the loss landscape, but he has *absolutely no idea* in which direction he should take a step to move himself closer to the bottom of the basin.

What is Chad to do? ***The answer is to apply gradient descent.*** All Chad needs to do is follow the slope of the gradient $\boldsymbol{W}$. We can compute the gradient $\boldsymbol{W}$ across all dimensions using the following equation:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{9.1}$$

In > 1 dimensions, our gradient becomes a *vector of partial derivatives*. The problem with this equation is that:

1. It's an *approximation* to the gradient.
2. It's painfully slow.

In practice, we use the *analytic gradient* instead. The method is exact and fast, but extremely challenging to implement due to partial derivatives and multi-variable calculus. Full derivation of the multivariable calculus used to justify gradient descent is outside the scope of this book. If you are interested in learning more about numeric and analytic gradients, I would suggest this lecture by Zibulevsky [80], Andrew Ng's cs229 machine learning notes [81], as well as the cs231n notes [82].

For the sake of this discussion, simply internalize what gradient descent is: attempting to optimize our parameters for low loss and high classification accuracy via an iterative process of taking a step in the direction that minimizes loss.

Treat It Like a Convex Problem (Even if It's Not)

Using the a bowl in Figure 9.1 (*right*) as a visualization of the loss landscape also allows us to draw an important conclusion in modern day neural networks – **we are treating the loss landscape as a convex problem, *even if it's not*.** If some function F is convex, then all local minima are also global minima. This idea fits the visualization of the bowl nicely. Our optimization algorithm simply has to strap on a pair of skis at the top of the bowl, then slowly ride down the gradient until we reach the bottom.

The issue is that nearly all problems we apply neural networks and deep learning algorithms to are *not* neat, convex functions. Instead, inside this bowl we'll find spike-like peaks, valleys that are more akin to canyons, steep dropoffs, and even slots where loss drops dramatically only to sharply rise again.

Given the non-convex nature of our datasets, why do we apply gradient descent? The answer is simple: *because it does a good enough job*. To quote Goodfellow et al. [10]:

> *"[An] optimization algorithm may not be guaranteed to arrive at even a local minimum in a reasonable amount of time, but it often finds a very low value of the [loss] function quickly enough to be useful."*

We can set the high expectation of finding a local/global minimum when training a deep learning network, but this expectation rarely aligns with reality. Instead, we end up finding a region of low loss – *this area may not even be a local minimum*, but in practice, it turns out that this is **good enough**.

The Bias Trick

Before we move on to implementing gradient descent, I want to take the time to discuss a technique called the "bias trick", a method of combining our weight matrix $\boldsymbol{W}$ and bias vector $\boldsymbol{b}$ into a *single* parameter. Recall from our previous decisions that our scoring function is defined as:

$$f(x_i, W, b) = Wx_i + b \tag{9.2}$$

It's often tedious to keep track of two separate variables, both in terms of explanation *and* implementation – to avoid situation this entirely, we can combine $\boldsymbol{W}$ and $\boldsymbol{b}$ together. To combine both the bias and weight matrix, we add an extra dimension (i.e., column) to our input data $\boldsymbol{X}$ that holds a constant 1 – this is our bias dimension.

Typically we either append the new dimension to each individual x_i as the first dimension or the last dimension. In reality, it doesn't matter. We can choose any arbitrary location to insert a