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 W and bias vector 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.

9.1 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.

9.1.1 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*):

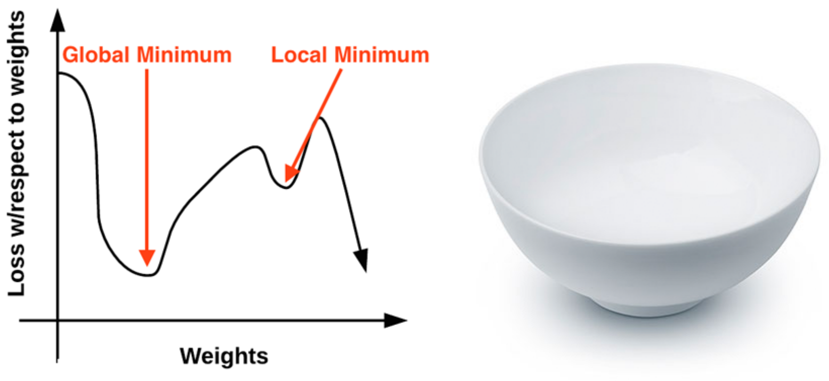

Figure 9.1: **Left:** The "naive loss" visualized as a 2D plot. **Right:** A more realistic loss landscape can be visualized as a bowl that exists in multiple dimensions. Our goal is to apply gradient descent to navigate to the bottom of this bowl (where there is low loss).

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 dimensions*, 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.

9.1.2 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 [85], Andrew Ng's cs229 machine learning notes [86], as well as the cs231n notes [87].

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.

9.1.3 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**.

9.1.4 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 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.

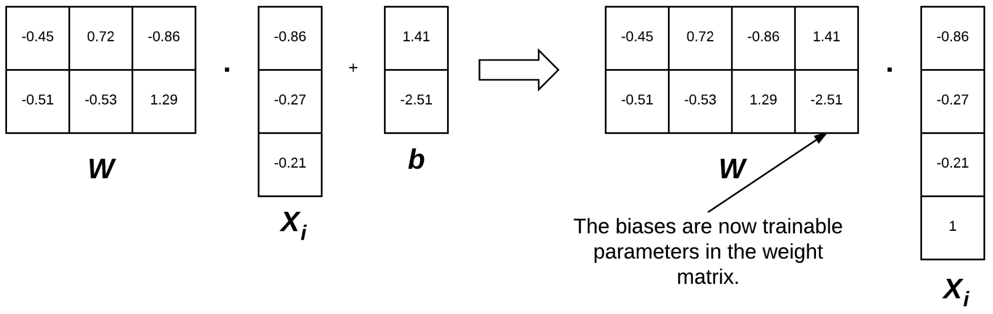

Figure 9.3: **Left:** Normally we treat the weight matrix and bias vector as two separate parameters. **Right:** However, we can actually *embed* the bias vector into the weight matrix (thereby making it a trainable parameter *directly inside* the weight matrix by initializing our input data $\boldsymbol{X}$ 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 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.

R We are actually inserting a new *row* in our feature vector in Figure 9.3 with a value of 1. In our implementation in Section 9.1.6 you will see that we add a *column* to our feature matrix filled with ones. So, which is correct? It actually depends on how you perform your linear algebra and how you are transposing each matrix. You will see both used in implementation and I want to ensure you are prepared for that now.

9.1.5 Pseudocode for Gradient Descent

Below I have included Python-like pseudocode for the standard, vanilla gradient descent algorithm (pseudocode inspired by cs231n slides [88]):

```
1 while True:
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 bouncing 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 spend a considerable amount of your time trying to find an optimal value for this variable for your model and dataset.

9.1.6 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:

```python
# import the necessary packages
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
import numpy as np
import argparse

def sigmoid_activation(x):
    # compute the sigmoid activation value for a given input
    return 1.0 / (1 + np.exp(-x))

def sigmoid_deriv(x):
    # compute the derivative of the sigmoid function ASSUMING
    # that the input ‘x‘ has already been passed through the sigmoid
    # activation function
    return x * (1 - 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`.

Lines 13-17 define the *derivative* of the sigmoid function. We need to compute the derivative of this function to derive the actual gradient. The gradient is what enables us to travel down the slope of the optimization surface. We will discuss this topic more in Chapter 10, so if you are new to computing the derivative of an activation function, don’t worry — this is simply an example. We’ll be covering it in more detail in our next chapter.

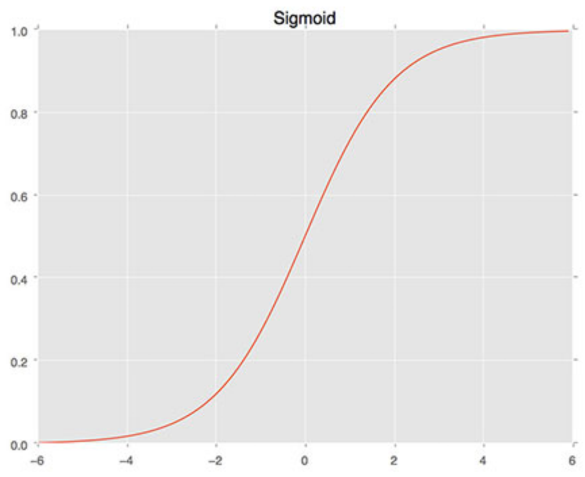

Figure 9.4: The sigmoid activation function. This function is centered at $x = 0.5, y = 0.5$. The function saturates at the tails.

The `predict` function applies our sigmoid activation function and then thresholds it based on whether the neuron is firing (1) or not (0):

```
19 def predict(X, W):
20     # take the dot product between our features and weight matrix
21     preds = sigmoid_activation(X.dot(W))
22 
23     # apply a step function to threshold the outputs to binary
24     # class labels
25     preds[preds <= 0.5] = 0
26     preds[preds > 0] = 1
27 
28     # return the predictions
29     return 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 21**). 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 25 and 26**). The predictions are then returned to the calling function on **Line 29**.

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:

```
31 # construct the argument parse and parse the arguments
32 ap = argparse.ArgumentParser()
33 ap.add_argument("-e", "--epochs", type=float, default=100,
34     help="# of epochs")
35 ap.add_argument("-a", "--alpha", type=float, default=0.01,
36     help="learning rate")
37 args = vars(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:

```
39 # generate a 2-class classification problem with 1,000 data points,
40 # where each data point is a 2D feature vector
41 (X, y) = make_blobs(n_samples=1000, n_features=2, centers=2,
42     cluster_std=1.5, random_state=1)
43 y = y.reshape((y.shape[0], 1))
44
45 # insert a column of 1's as the last entry in the feature
46 # matrix -- this little trick allows us to treat the bias
47 # as a trainable parameter within the weight matrix
48 X = np.c_[X, np.ones((X.shape[0]))]
49
50 # partition the data into training and testing splits using 50% of
51 # the data for training and the remaining 50% for testing
52 (trainX, testX, trainY, testY) = train_test_split(X, y,
53     test_size=0.5, random_state=42)
```

On **Line 41** 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 48 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 52 and 53**, 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):

```
55 # initialize our weight matrix and list of losses
56 print("[INFO] training...")
57 W = np.random.randn(X.shape[1], 1)
58 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 [89].

Line 58 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:

```
60 # loop over the desired number of epochs
61 for epoch in np.arange(0, args["epochs"]):
62     # take the dot product between our features ‘X‘ and the weight
63     # matrix ‘W‘, then pass this value through our sigmoid activation
64     # function, thereby giving us our predictions on the dataset
65     preds = sigmoid_activation(trainX.dot(W))
66 
67     # now that we have our predictions, we need to determine the
68     # ‘error‘, which is the difference between our predictions and
69     # the true values
70     error = preds - trainY
71     loss = np.sum(error ** 2)
72     losses.append(loss)
```

On **Line 61** 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 65 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 70**). **Line 71** 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 72**, 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`:

```
74     # the gradient descent update is the dot product between our
75     # (1) features and (2) the error of the sigmoid derivative of
76     # our predictions
77     d = error * sigmoid_deriv(preds)
78     gradient = trainX.T.dot(d)
```

```
79
80      # in the update stage, all we need to do is "nudge" the weight
81      # matrix in the negative direction of the gradient (hence the
82      # term "gradient descent" by taking a small step towards a set
83      # of "more optimal" parameters
84      W += -args["alpha"] * gradient
85
86      # check to see if an update should be displayed
87      if epoch == 0 or (epoch + 1) % 5 == 0:
88          print("[INFO] epoch={}, loss={:.7f}".format(int(epoch + 1),
89              loss))
```

Line 77 and 78 handles computing the gradient, which is the dot product between our data points and the error multiplied by the sigmoid derivative of our predictions.

Line 84 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 87-89**) 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:

```
91  # evaluate our model
92  print("[INFO] evaluating...")
93  preds = predict(testX, W)
94  print(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 94**.

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:

```
96  # plot the (testing) classification data
97  plt.style.use("ggplot")
98  plt.figure()
99  plt.title("Data")
100 plt.scatter(testX[:, 0], testX[:, 1], marker="o", c=testY[:, 0], s=30)
101
102 # construct a figure that plots the loss over time
103 plt.style.use("ggplot")
104 plt.figure()
105 plt.plot(np.arange(0, args["epochs"]), losses)
106 plt.title("Training Loss")
107 plt.xlabel("Epoch #")
108 plt.ylabel("Loss")
109 plt.show()
```

9.1.7 Simple Gradient Descent Results

To execute our script, simply issue the following command:

```
$ python gradient_descent.py
[INFO] training...
[INFO] epoch=1, loss=194.3629849
[INFO] epoch=5, loss=9.3225755
[INFO] epoch=10, loss=5.2176352
[INFO] epoch=15, loss=3.0483912
[INFO] epoch=20, loss=1.8903512
[INFO] epoch=25, loss=1.3532867
[INFO] epoch=30, loss=1.0746259
[INFO] epoch=35, loss=0.9074719
[INFO] epoch=40, loss=0.7956885
[INFO] epoch=45, loss=0.7152976
[INFO] epoch=50, loss=0.6547454
[INFO] epoch=55, loss=0.6078759
[INFO] epoch=60, loss=0.5711027
[INFO] epoch=65, loss=0.5421195
[INFO] epoch=70, loss=0.5192554
[INFO] epoch=75, loss=0.5011559
[INFO] epoch=80, loss=0.4866579
[INFO] epoch=85, loss=0.4747733
[INFO] epoch=90, loss=0.4647116
[INFO] epoch=95, loss=0.4558868
[INFO] epoch=100, loss=0.4478966
```

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 > 194 but drops to ≈ 0.6 by epoch 50. By the time training terminates by epoch 100, our loss has to 0.045.

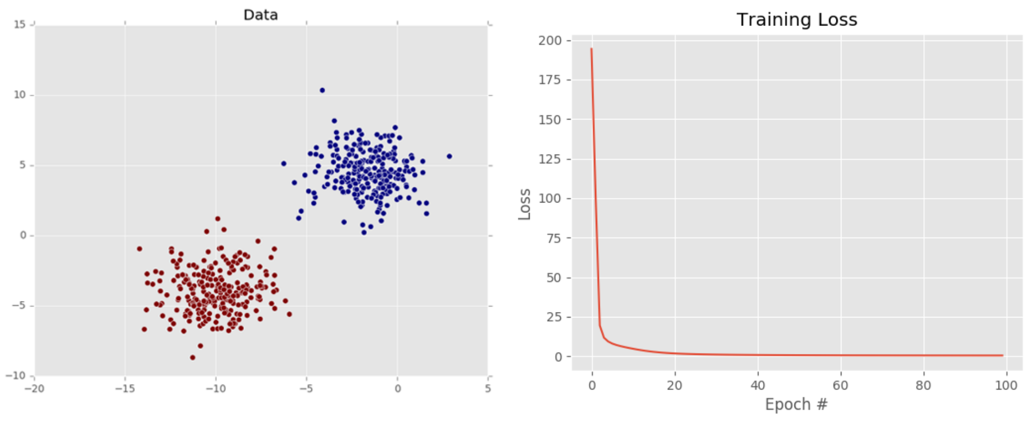

Figure 9.5: **Left:** The input dataset that we are trying to classify into two sets: red and blue. This dataset is clearly linearly separable as we can draw a single line that neatly divides the dataset into two classes. **Right:** Learning a set of parameters to classify our dataset via gradient descent. 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. We can see the classification report for our dataset below:

```
[INFO] evaluating...
             precision    recall  f1-score   support

          0       1.00      1.00      1.00       250
          1       1.00      1.00      1.00       250

avg / total       1.00      1.00      1.00       500
```

Notice how *both* classes are correctly classified 100% of the time, again, implying that our dataset is both: (1) linearly separable and (2) our gradient descent algorithm was able to descend into an area of low loss, capable of separating the two classes.

That said, it's important to keep in mind how the vanilla gradient descent algorithm works. 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 may not be able to learn a model that can separate the points (even though they are linearly separable).

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.

9.2 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* 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 effects 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 [90], 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.

9.2.1 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:

```
while True:
    batch = next_training_batch(data, 256)
    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 (http://pyimg.co/pd5w0), 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.

9.2.2 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  # import the necessary packages
2  from sklearn.model_selection import train_test_split
3  from sklearn.metrics import classification_report
4  from sklearn.datasets import make_blobs
5  import matplotlib.pyplot as plt
6  import numpy as np
7  import argparse
8
9  def sigmoid_activation(x):
10     # compute the sigmoid activation value for a given input
11     return 1.0 / (1 + np.exp(-x))
12
13 def sigmoid_deriv(x):
14     # compute the derivative of the sigmoid function ASSUMING
15     # that the input ‘x‘ has already been passed through the sigmoid
16     # activation function
17     return x * (1 - x)
```

Lines 2-7 import our required Python packages, exactly the same as the `gradient_descent.py` example earlier in this chapter. **Lines 9-17** defines our `sigmoid_activation` and `sigmoid_deriv` functions, both of which are identical to the previous version of gradient descent.

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

```
19 def predict(X, W):
20     # take the dot product between our features and weight matrix
21     preds = sigmoid_activation(X.dot(W))
22 
23     # apply a step function to threshold the outputs to binary
24     # class labels
25     preds[preds <= 0.5] = 0
26     preds[preds > 0] = 1
27 
28     # return the predictions
29     return preds
```

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

```
31 def next_batch(X, y, batchSize):
32     # loop over our dataset `X` in mini-batches, yielding a tuple of
33     # the current batched data and labels
34     for i in np.arange(0, X.shape[0], batchSize):
35         yield (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 34 and 35 then loop over the training examples, yielding subsets of both `X` and `y` as mini-batches.

Next, we can parse our command line arguments:

```
37 # construct the argument parse and parse the arguments
38 ap = argparse.ArgumentParser()
39 ap.add_argument("-e", "--epochs", type=float, default=100,
40     help="# of epochs")
41 ap.add_argument("-a", "--alpha", type=float, default=0.01,
42     help="learning rate")
43 ap.add_argument("-b", "--batch-size", type=int, default=32,
44     help="size of SGD mini-batches")
45 args = vars(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:

```
47 # generate a 2-class classification problem with 1,000 data points,
48 # where each data point is a 2D feature vector
49 (X, y) = make_blobs(n_samples=1000, n_features=2, centers=2,
50     cluster_std=1.5, random_state=1)
51 y = y.reshape((y.shape[0], 1))
```

```
52
53 # insert a column of 1's as the last entry in the feature
54 # matrix -- this little trick allows us to treat the bias
55 # as a trainable parameter within the weight matrix
56 X = np.c_[X, np.ones((X.shape[0]))]
57
58 # partition the data into training and testing splits using 50% of
59 # the data for training and the remaining 50% for testing
60 (trainX, testX, trainY, testY) = train_test_split(X, y,
61     test_size=0.5, random_state=42)
```

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

```
63 # initialize our weight matrix and list of losses
64 print("[INFO] training...")
65 W = np.random.randn(X.shape[1], 1)
66 losses = []
```

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

```
68 # loop over the desired number of epochs
69 for epoch in np.arange(0, args["epochs"]):
70     # initialize the total loss for the epoch
71     epochLoss = []
72
73     # loop over our data in batches
74     for (batchX, batchY) in next_batch(trainX, trainY, args["batch_size"]):
75         # take the dot product between our current batch of features
76         # and the weight matrix, then pass this value through our
77         # activation function
78         preds = sigmoid_activation(batchX.dot(W))
79
80         # now that we have our predictions, we need to determine the
81         # `error`, which is the difference between our predictions
82         # and the true values
83         error = preds - batchY
84         epochLoss.append(np.sum(error ** 2))
```

On **Line 69** we start looping over the supplied number of `--epochs`. We then loop over our training data in batches on **Line 74**. 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 error for the batch on **Line 83** and use this value to update our least squares `epochLoss` on **Line 84**.

Now that we have the `error`, we can compute the gradient descent update, identical to computing the gradient from vanilla gradient descent, only this time we are performing the update on *batches* rather than the *entire* training set:

```
86         # the gradient descent update is the dot product between our
87         # (1) current batch and (2) the error of the sigmoid
88         # derivative of our predictions
```

```
89            d = error * sigmoid_deriv(preds)
90            gradient = batchX.T.dot(d)
91
92            # in the update stage, all we need to do is "nudge" the
93            # weight matrix in the negative direction of the gradient
94            # (hence the term "gradient descent" by taking a small step
95            # towards a set of "more optimal" parameters
96            W += -args["alpha"] * gradient
```

Line 96 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:

```
98        # update our loss history by taking the average loss across all
99        # batches
100       loss = np.average(epochLoss)
101       losses.append(loss)
102
103       # check to see if an update should be displayed
104       if epoch == 0 or (epoch + 1) % 5 == 0:
105           print("[INFO] epoch={}, loss={:.7f}".format(int(epoch + 1),
106               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:

```
108   # evaluate our model
109   print("[INFO] evaluating...")
110   preds = predict(testX, W)
111   print(classification_report(testY, preds))
```

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

```
113   # plot the (testing) classification data
114   plt.style.use("ggplot")
115   plt.figure()
116   plt.title("Data")
117   plt.scatter(testX[:, 0], testX[:, 1], marker="o", c=testY[:, 0], s=30)
118
119   # construct a figure that plots the loss over time
120   plt.style.use("ggplot")
121   plt.figure()
122   plt.plot(np.arange(0, args["epochs"]), losses)
123   plt.title("Training Loss")
124   plt.xlabel("Epoch #")
125   plt.ylabel("Loss")
126   plt.show()
```

9.2.3 SGD Results

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

```
$ python sgd.py
[INFO] training...
[INFO] epoch=1, loss=0.1317637
[INFO] epoch=5, loss=0.0162487
[INFO] epoch=10, loss=0.0112798
[INFO] epoch=15, loss=0.0100234
[INFO] epoch=20, loss=0.0094581
[INFO] epoch=25, loss=0.0091053
[INFO] epoch=30, loss=0.0088366
[INFO] epoch=35, loss=0.0086082
[INFO] epoch=40, loss=0.0084031
[INFO] epoch=45, loss=0.0082138
[INFO] epoch=50, loss=0.0080364
[INFO] epoch=55, loss=0.0078690
[INFO] epoch=60, loss=0.0077102
[INFO] epoch=65, loss=0.0075593
[INFO] epoch=70, loss=0.0074153
[INFO] epoch=75, loss=0.0072779
[INFO] epoch=80, loss=0.0071465
[INFO] epoch=85, loss=0.0070207
[INFO] epoch=90, loss=0.0069001
[INFO] epoch=95, loss=0.0067843
[INFO] epoch=100, loss=0.0066731
[INFO] evaluating...
             precision    recall  f1-score   support

          0       1.00      1.00      1.00       250
          1       1.00      1.00      1.00       250

avg / total       1.00      1.00      1.00       500
```

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. The results of which can be seen in Figure 9.6.

Investigating the actual loss values at the end of the 100th epoch, you'll notice that loss obtained by SGD is ***nearly two orders of magnitude lower*** than vanilla gradient descent (0.006 vs 0.447, 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.

9.3 Extensions to SGD

There are two primary extensions that you'll encounter to SGD in practice. The first is momentum [91], 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 [92], an extension to standard momentum.

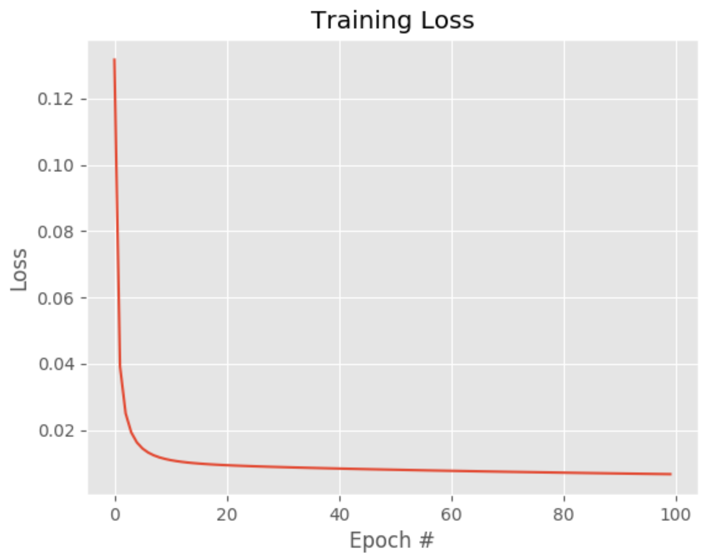

Figure 9.6: Applying Stochastic Gradient Descent to our dataset of red and blue data points from Figure 9.5. We are able to nearly two orders of magnitude lower loss by the end of the 100th epoch (as compared to standard, vanilla gradient descent) due to the multiple weight updates per batch.

9.3.1 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 [91, 93].

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

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

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

$$V = \gamma V_{t-1} + \alpha \nabla_{\boldsymbol{W}} f(W) \qquad W = W - V_t \tag{9.5}$$

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 [94] and Qian [91].

9.3.2 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 [92] 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 [95], we can see a nice visualization of Nesterov acceleration (Figure 9.7).

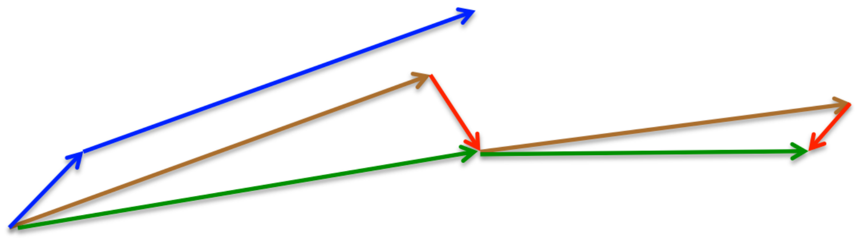

Figure 9.7: A graphical depecition of Nesterov acceleration. First, we make a big jump in the direction of the previous gradient, then measure the gradient where we ended up and make the correction.

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 [93]).

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 [93], Bengio [96], and Sutskever [97].

9.3.3 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) at 0.9 although Karpathy [98] 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 [99], VGGNet [100], ResNet [101], Inception [102], 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.

9.4 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 [103], 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.

9.4.1 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).

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 (orange). 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

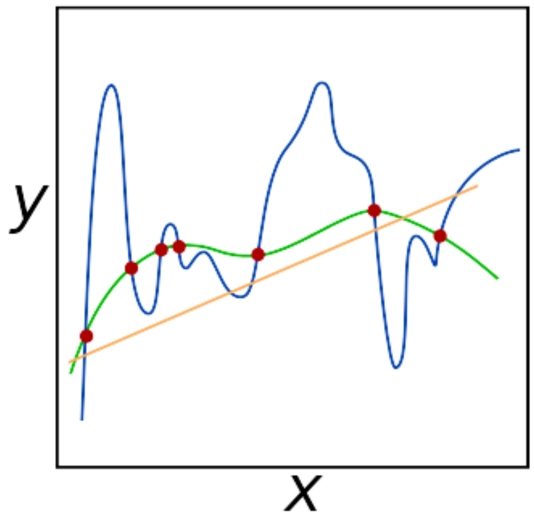

Figure 9.8: An example of underfitting (orange line), overfitting (blue line), and generalizing (green line). Our goal when building deep learning classifiers is to obtain these types of "green functions" that fit our training data nicely, but avoid overfitting. Regularization can help us obtain this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

regularization and the impact it has on our loss function and weight update rule, let's proceed to the next section.

9.4.2 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 = 0

for i in np.arange(0, W.shape[0]):
    for j in np.arange(0, W.shape[1]):
        penalty += (W[i][j] ** 2)
```

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 the effect 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 spend 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:

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

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

$$W = W - \alpha\nabla_{\boldsymbol{W}} f(W) - \lambda R(W) \tag{9.13}$$

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.

9.4.3 Types of Regularization Techniques

In general, you'll see three common types of regularization that 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 [103] 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 DeepLearning.net tutorial [104], and the notes from Karpathy's cs231n Neural Networks II lecture [105].

9.4.4 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:

```python
# import the necessary packages
from sklearn.linear_model import SGDClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from pyimagesearch.preprocessing import SimplePreprocessor
from pyimagesearch.datasets import SimpleDatasetLoader
from imutils import paths
import 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:

```
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-d", "--dataset", required=True,
     help="path to input dataset")
args = vars(ap.parse_args())

# grab the list of image paths
print("[INFO] loading images...")
imagePaths = list(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:

```
# initialize the image preprocessor, load the dataset from disk,
# and reshape the data matrix
sp = SimplePreprocessor(32, 32)
sdl = SimpleDatasetLoader(preprocessors=[sp])
(data, labels) = sdl.load(imagePaths, verbose=500)
data = data.reshape((data.shape[0], 3072))
```

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:

```
# encode the labels as integers
le = LabelEncoder()
labels = le.fit_transform(labels)

# partition the data into training and testing splits using 75% of
# the data for training and the remaining 25% for testing
(trainX, testX, trainY, testY) = train_test_split(data, labels,
     test_size=0.25, random_state=5)
```

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

```
# loop over our set of regularizers
for r in (None, "l1", "l2"):
     # train a SGD classifier using a softmax loss function and the
     # specified regularization function for 10 epochs
     print("[INFO] training model with ‘{}‘ penalty".format(r))
     model = SGDClassifier(loss="log", penalty=r, max_iter=10,
42          learning_rate="constant", eta0=0.01, random_state=42)
43      model.fit(trainX, trainY)
44
45      # evaluate the classifier
46      acc = model.score(testX, testY)
47      print("[INFO] ‘{}‘ penalty accuracy: {:.2f}%".format(r,
48          acc * 100))
```

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
[INFO] loading images...
...
[INFO] training model with ‘None‘ penalty
[INFO] ‘None‘ penalty accuracy: 50.40%
[INFO] training model with ‘l1‘ penalty
[INFO] ‘l1‘ penalty accuracy: 52.53%
[INFO] training model with ‘l2‘ penalty
[INFO] ‘l2‘ penalty accuracy: 55.07%
```

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%**.

R 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 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*.

9.5 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.