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 [76], FLANN [71], or Annoy [77], 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) [78]

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.

R Much of this chapter was inspired by Andrej Karpathy's excellent Linear Classification notes inside Stanford's cs231n class [79]. A big thank you to Karpathy and the rest of the cs231n teaching assistants for putting together such accessible notes.

8.1 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.

8.1.1 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 $X \subseteq R^{100\times(32\times32\times3)}$ 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.

8.1.2 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.

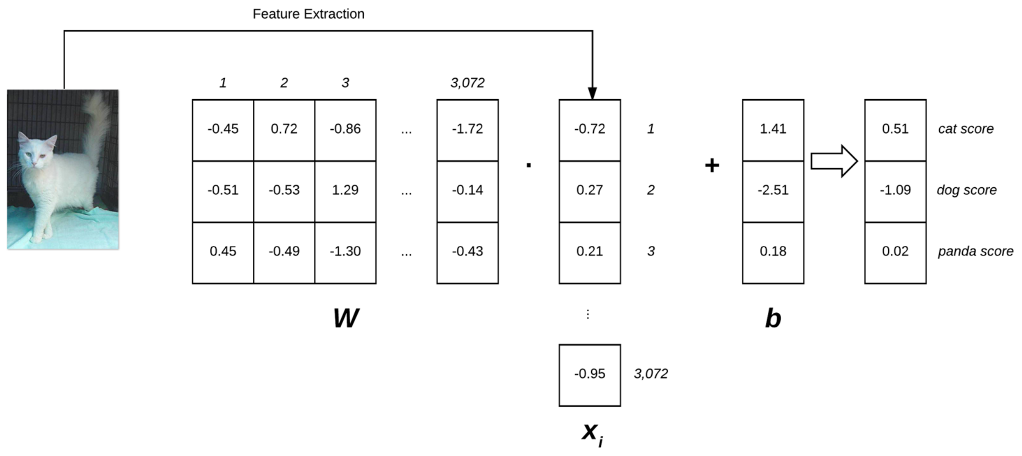

Figure 8.1: Illustrating the dot product of the weight matrix $\boldsymbol{W}$ and feature vector $\boldsymbol{x}$, followed by the addition of the bias term. Figure inspired by Karpathy's example in Stanford University's cs231n course [57].

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.

R Readers who are unfamiliar with taking dot products should read this quick and concise tutorial: http://pyimg.co/fgcvp. 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 [80].

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.

8.1.3 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 $\boldsymbol{W}$ and the bias vector $\boldsymbol{b}$.** 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.

8.1.4 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  # import the necessary packages
2  import numpy as np
3  import cv2
4
5  # initialize the class labels and set the seed of the pseudorandom
6  # number generator so we can reproduce our results
7  labels = ["dog", "cat", "panda"]
8  np.random.seed(1)
```

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 # randomly initialize our weight matrix and bias vector -- in a
11 # *real* training and classification task, these parameters would
```

```
12 # be *learned* by our model, but for the sake of this example,
13 # let’s use random values
14 W = np.random.randn(3, 3072)
15 b = np.random.randn(3)
```

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 # load our example image, resize it, and then flatten it into our
18 # "feature vector" representation
19 orig = cv2.imread("beagle.png")
20 image = cv2.resize(orig, (32, 32)).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 # compute the output scores by taking the dot product between the
23 # weight matrix and image pixels, followed by adding in the bias
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 # loop over the scores + labels and display them
27 for (label, score) in zip(labels, scores):
28     print("[INFO] {}: {:.2f}".format(label, score))
29
30 # draw the label with the highest score on the image as our
31 # prediction
32 cv2.putText(orig, "Label: {}".format(labels[np.argmax(scores)]),
33     (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 255, 0), 2)
34
```

```
35 # display our input image
36 cv2.imshow("Image", orig)
37 cv2.waitKey(0)
```

To execute our example, just issue the following command:

```
$ python linear_example.py
[INFO] dog: 7963.93
[INFO] cat: -2930.99
[INFO] panda: 3362.47
```

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.

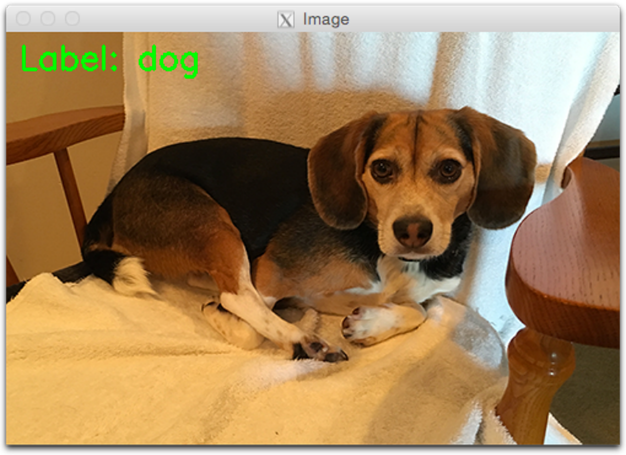

Figure 8.3: In this example, our linear classifier was correctly able to label the input image as *dog*; however, keep in mind that this is a worked example. Later in this book you'll learn how to *train* our weights and biases to automatically make these predictions.

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.

8.2 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 [81], Witten et al. [82], Harrington [83], and Marsland [84] if you would like to complement this chapter with more mathematically rigorous derivations.

8.2.1 What Are Loss Functions?

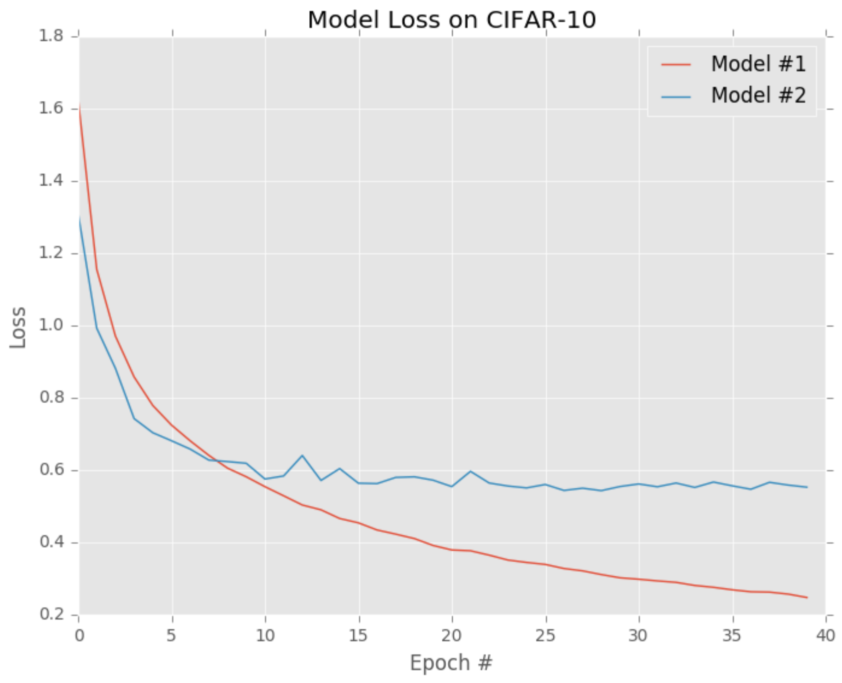

Figure 8.4: The training losses for two separate models trained on the CIFAR-10 dataset are plotted over time. Our loss function quantifies how "good" or "bad" of a job a given model is doing at classifying data points from the dataset. Model #1 achieves considerably lower loss than Model #2.

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 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.

8.2.2 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***:

$$L_i = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1) \tag{8.5}$$

R 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:

```
>>> max(0, 1.33 - 4.26 + 1) + max(0, -1.01 - 4.26 + 1)
0
```

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:

```
>>> max(0, 3.76 - (-1.20) + 1) + max(0, -3.81 - (-1.20) + 1)
5.96
```

	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:

```
>>> max(0, -2.37 - (-2.27) + 1) + max(0, 1.03 - (-2.27) + 1)
5.199999999999999
```

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:

```
>>> (0.0 + 5.96 + 5.2) / 3.0
3.72
```

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.

8.2.3 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

	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 of Equation 8.11, 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.

R 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.

8.3 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, 79].

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.