Learning Signal and Ignoring Noise
Introduction to Regularization & Batching

8

IN THIS CHAPTER

Overfitting

Dropout

Batch Gradient Descent

“This is a sample quote that I'm trying to make, but Andrew... would you like to perhaps put something else here?”

— *SOME AUTHOR*

3 Layer Network on MNIST

Let's return to our MNIST dataset and attempt to classify it with our new network

In last several chapters, we have learned that neural networks model correlation. In fact, the hidden layers (the middle one in our 3 layer network) can even create "intermediate" correlation to help solve for a task (seemingly out of mid air). How do we know that our network is creating good correlation?

Back when we learned about Stochastic Gradient Descent with Multiple Inputs, we ran an experiment where we froze one weight and then asked our network to continue training. As it was training, we watch the "dots find the bottom of the bowls" as it were. We were watching the weights become adjusted to minimize the error.

When we froze the weight, however, we were surprised to see that the frozen weight still found the bottom of the bowl! For some reason, the bowl moved so that the frozen weight value becamse optimal. Furthemore, if we unfroze the weight after training to do some more training, it wouldn't learn! Why? Well, the error had already fallen to 0! As far as the network was concerned, there was nothing more to learn!

This begs the question, what if the input to the frozen weight was actually important to predicting baseball victory in the real world? What if the network had figured out a way to accurately predict the games in the training dataset (because that's what networks do, they minimize the error), but somehow forgot to include a valuable input?

This phenomenon is, unfortunately, extremely common in neural networks. In fact, on might say it's "The Arch Nemesis" of neural networks, **Overfitting**, and unfortunately, the more powerful your neural network's expressive power (more layers / weights), the more prone the network is to overfit. So, there's an everlasting battle going on in research where people continually find tasks that need more powerful layers but find themselves having to do lots of problem solving to make sure the network doesn't "overfit".

In this chapter, we're going to study the basics of Regularization, which are key to combatting overfitting in neural networks. In order to do this, we're going first start with our most powerful neural network (3 layer network with relu hidden layer) on our most challenging task (MNIST digit classification).

So, to start, go ahead and train the network on the following page. You should see the same results as those listed below. Alas! Our network learned to perfectly predict the training data! Should we celebrate?

```
import numpy as np

np.random.seed(1)

def relu(x):
    return (x >= 0) * x # returns x if x > 0
                       # return 0 otherwise

def relu2deriv(output):
    return output >= 0 # returns 1 for input > 0
                    # return 0 otherwise

alpha = 0.005
iterations = 300
hidden_size = 40
pixels_per_image = 784
num_labels = 10

weights_0_1 = 0.2*np.random.random((pixels_per_image,hidden_size)) - 0.1
weights_1_2 = 0.2*np.random.random((hidden_size,num_labels)) - 0.1

for j in xrange(iterations):

    error = 0.0
    correct_cnt = 0

    for i in xrange(len(images)):

        layer_0 = images[i:i+1]
        layer_1 = relu(np.dot(layer_0,weights_0_1))
        layer_2 = np.dot(layer_1,weights_1_2)

        error += np.sum((labels[i:i+1] - layer_2) ** 2)

        correct_cnt += int(np.argmax(layer_2) == \
                                        np.argmax(labels[i:i+1]))

        layer_2_delta = (labels[i:i+1] - layer_2)
        layer_1_delta = layer_2_delta.dot(weights_1_2.T)\
                                    * relu2deriv(layer_1)

        weights_1_2 += alpha * layer_1.T.dot(layer_2_delta)
        weights_0_1 += alpha * layer_0.T.dot(layer_1_delta)


    sys.stdout.write("\r"+ \
                    " I:"+str(j)+ \
                    " Error:" + str(error/float(len(images)))[0:5] +\
                    " Correct:" + str(correct_cnt/float(len(images))))
```

```
....
I:299 Error:0.101 Correct:1.0
```

Well... that was easy!

Our neural network perfectly learned to predict all 1000 images!

So, in some ways, this is a real victory. Our neural network was able to take a dataset of 1000 images and learn to correlate each input image with the correct label. How did it do this? Well, it simply iterated through each image, made a prediction, and then updated each weight ever so slightly so that the prediction was better next time. Doing this long enough on all the images eventually reached a state where the network could correctly predict on all of the images!

Perhaps a non-obvious question: how well will the neural network do on an image that it hasn't seen before? In other words, how well will it do on an image that wasn't part of the 1000 images it was trained on? Well, our MNIST dataset has many more images than just the 1000 we trained on; let's try it out! In the same notebook from before "Chapter 6 Neural Networks in the Real World" there are two variables called images_test and labels_test. If you execute the following code, it will run the neural network on these images and evaluate how well the network classifies them.

```
error = 0.0
correct_cnt = 0

for i in xrange(len(test_images)):

    layer_0 = test_images[i:i+1]
    layer_1 = relu(np.dot(layer_0,weights_0_1))
    layer_2 = np.dot(layer_1,weights_1_2)

    error += np.sum((test_labels[i:i+1] - layer_2) ** 2)

    correct_cnt += int(np.argmax(layer_2) == \
                                    np.argmax(test_labels[i:i+1]))
sys.stdout.write("\r"+ \
                 " Error:" + str(error/float(len(test_images)))[0:5] +\
                 " Correct:" + str(correct_cnt/float(len(test_images))))
```

```
Error:0.803 Correct:0.601
```

The network did horribly! It only predicted with an accuracy of 60.1%. Why does it do so terribly on these new testing images when it learned to predict with 100% accuracy on the training data? How strange! We call this 60.1% number the *test accuracy*. It's the accuracy of the neural network on data that the network was NOT trained on. This number is important because it simulates how well our neural network will perform if we tried to use it in the real world (which only gives us images we haven't seen before). This is the score that matters!

Memorization vs Generalization

"Memorizing" 1000 images is easier than "Generalizing" to all images.

Let's consider again how the neural network learns. It adjusts each weight in each matrix so that the network is better able to take *specific inputs* and make a *specific prediction*. Perhaps a better question might be, "If we train it on 1000 images which it learns to predict perfectly, why does it work on other images at all?" As you might expect, when our fully trained neural network is applied to a new image, it is only guaranteed to work well if the new image is *nearly identical to an image from the training data*. Why? Well, the neural network only learned to transform input data to output data for *very specific input configurations*. If you give it something that doesn't look *familiar*, then it will predict randomly!

Well this makes neural networks kindof pointless! What's the point in a neural network only working on the data you trained it on! You already know the correct classifications for those datapoints! Neural networks are only useful if they work on data we don't already know the answer to! As it turns out, there's a way to combat this. For a clue, below I've printed out BOTH the training *and testing accuracy* of the neural network *as it was training* (every 10 iterations). Notice anything interesting? You should see a clue to better networks!

```
I:0 Test-Err:0.650 Test-Acc:0.595 Train-Err:0.743 Train-Acc:0.511
I:10 Test-Err:0.471 Test-Acc:0.746 Train-Err:0.293 Train-Acc:0.917
I:20 Test-Err:0.478 Test-Acc:0.753 Train-Err:0.222 Train-Acc:0.952
I:30 Test-Err:0.501 Test-Acc:0.74 Train-Err:0.189 Train-Acc:0.974
I:40 Test-Err:0.527 Test-Acc:0.728 Train-Err:0.170 Train-Acc:0.986
I:50 Test-Err:0.552 Test-Acc:0.708 Train-Err:0.158 Train-Acc:0.991
I:60 Test-Err:0.574 Test-Acc:0.692 Train-Err:0.150 Train-Acc:0.992
I:70 Test-Err:0.595 Test-Acc:0.676 Train-Err:0.144 Train-Acc:0.993
I:80 Test-Err:0.614 Test-Acc:0.661 Train-Err:0.138 Train-Acc:0.995
I:90 Test-Err:0.633 Test-Acc:0.651 Train-Err:0.134 Train-Acc:0.996
I:100 Test-Err:0.650 Test-Acc:0.645 Train-Err:0.129 Train-Acc:0.996
I:110 Test-Err:0.667 Test-Acc:0.635 Train-Err:0.126 Train-Acc:0.997
I:120 Test-Err:0.682 Test-Acc:0.633 Train-Err:0.123 Train-Acc:0.998
I:130 Test-Err:0.697 Test-Acc:0.631 Train-Err:0.121 Train-Acc:0.998
I:140 Test-Err:0.711 Test-Acc:0.627 Train-Err:0.119 Train-Acc:0.999
I:150 Test-Err:0.722 Test-Acc:0.627 Train-Err:0.117 Train-Acc:0.999
I:160 Test-Err:0.733 Test-Acc:0.625 Train-Err:0.116 Train-Acc:0.999
I:170 Test-Err:0.742 Test-Acc:0.62 Train-Err:0.114 Train-Acc:0.999
I:180 Test-Err:0.750 Test-Acc:0.616 Train-Err:0.112 Train-Acc:0.999
I:190 Test-Err:0.758 Test-Acc:0.614 Train-Err:0.111 Train-Acc:0.999
I:200 Test-Err:0.765 Test-Acc:0.612 Train-Err:0.110 Train-Acc:0.999
I:210 Test-Err:0.771 Test-Acc:0.611 Train-Err:0.108 Train-Acc:0.999
I:220 Test-Err:0.776 Test-Acc:0.612 Train-Err:0.107 Train-Acc:0.999
I:230 Test-Err:0.779 Test-Acc:0.611 Train-Err:0.106 Train-Acc:0.999
I:240 Test-Err:0.783 Test-Acc:0.61 Train-Err:0.105 Train-Acc:0.999
I:250 Test-Err:0.786 Test-Acc:0.605 Train-Err:0.104 Train-Acc:0.999
I:260 Test-Err:0.790 Test-Acc:0.606 Train-Err:0.103 Train-Acc:0.999
I:270 Test-Err:0.793 Test-Acc:0.603 Train-Err:0.103 Train-Acc:0.999
I:280 Test-Err:0.796 Test-Acc:0.6 Train-Err:0.102 Train-Acc:0.999
I:290 Test-Err:0.800 Test-Acc:0.602 Train-Err:0.101 Train-Acc:1.0
```

Overfitting in Neural Networks

Neural networks can get worse if we train them too much?

For some reason, our *test* accuracy went up for the first 20 iterations, and then slowly decreased as the network trained more and more (during which time the *training* accuracy was still improving). This is very common in neural networks. Let me explain the phenomenon via analogy.

Imagine that you are creating a *mold* for a common dinner fork, but instead of using it to create other forks you want to use it to *identify* if a particular utensil is in fact, a fork. If an object fits in the mold, you would conclude that the object is a fork, and if it does not, then you would conclude that it is *not* a fork.

Let's say you set out to make this mold, and you start with a wet piece of clay and a big bucket of 3-pronged forks, spoons, and knives. You then press each of the forks into the same place in the mold to create an outline, which sortof looks like a mushy fork. You repeatedly place all the forks in the clay over and over, hundreds of times. When you let the clay dry, you then find that none of the spoons or knives fit into this mold, but all of the forks fit into this mold. Awesome! You did it! You correctly made a mold that can only fit the shape of a fork!

However, what happens if someone hands you a 4-pronged fork? You look in your mold and notice that there is a specific outlines for three, thin prongs in your clay. Your 4-pronged fork doesn't fit! Why not? It's still a fork!

This is because the clay wasn't molded on any 4-pronged forks. It was only molded on the on 3-pronged variety. In this way, the clay has **overfit** to only recognize the types of forks that it was "trained" to shape.

This is exactly the same phenomenon that we just witnessed in our neural new-tork. It's actually an even closer parallel than you might think. In truth, one way to view the weights of a neural network is as a *high dimensional shape*. As you train, this shape *molds* around the shape of your data, learning to distinguish one pattern from another. Unfortunately, the images in our testing dataset were *slightly* different than the patterns in our training dataset. This caused our network to fail on many of our *testing* examples.

This phenomenon is known as **Overfitting**. A more official definition of a neural network that overfits is a neural network that has learned the *noise* in the dataset instead of only making decisions based on the *true signal*.

Where Overfitting Comes From

What causes our neural networks to overfit?

Let's alter this scenario just a bit. Picture the fresh clay in your head again (unmolded). What if you only pushed a single fork into it? Assuming the clay was very thick, it wouldn't have as much detail as the previous mold did (which was in-printed many times) Thus, it would be only a *very general shape of a fork*. This shape might, in fact, be compatible with both the 3 and 4 pronged variety of fork, because it's still a very *fuzzy* imprint.

Assuming this information, our mold actually got worse at our testing dataset as we imprinted more forks because it learned more *detailed information* about the training dataset that it was being molded to. This caused it to reject images that were even the slightest bit off from what it had repeatedly seen in the training data. So, what is this *detailed information* in our images that is incompatible with our test data? In our fork analogy, this was the number of prongs on the fork. In images, it's generally referred to as *noise*. In reality, it's a bit more nuanced. Consider these two dog pictures.

Everything that makes these pictures unique *beyond* what captures the essence of "dog" is included in this term *noise*. In the picture on the left, the pillow and the background are both noise. In the picture on the right, the empty, middle blackness of the dog is actually a form of *noise* as well. It's really the edges that tell us that it's a dog. The middle blackness doesn't really tell us anything. On the picture on the left, however, the middle of the dog has the furry texture and color of a dog, which could help the classifier correctly identify it.

So, how do we get our neural networks to train only on the *signal* (the essence of a dog) and ignore the *noise* (other stuff irrelevant to the classification)? Well, one way of doing this is by **early stopping**. It turns out that a large amount of noise comes in the fine grained detail of an image, and most of the signal (for objects) is found in the general *shape* and perhaps *color* of the image.

The Simplest Regularization: Early Stopping

Stop training the network when it starts getting worse.

So, how do we get our neural networks to train only on the *signal* (the essence of a dog) and ignore the *noise* (other stuff irrelevant to the classification)? Well, one way of doing this is by **early stopping**. It turns out that a large amount of noise often comes in the fine grained detail present in input data, and most of the signal is found in the more general characteristics of your input data (for images, this is things like big shapes and color).

So, how do we get our neural network to ignore the fine grained detail and only capture the general information present in our data (i.e. the general shape of dog or of an MNIST digit)? Well, we don't let the network train long enough to learn it! Just like in the "fork mold" example, it takes many forks imprinted many times to create the perfect outline of a 3 pronged fork. The first few imprints only generally capture the shallow outline of a fork. The same can be said for neural networks. As a result, early stopping is the cheapest form of "regularization", and if you're in a pinch, it can be quite effective.

This brings us to the subject that this chapter is all about, **Regularization**. Regularization is a subfield of methods for getting your model to *generalize* to new data points (intead of just memorize the training data). It's a subset of methods that help your neural network learn the *signal* and ignore the *noise*. In our case, it's a toolset at our disposal to create neural networks that have these properties.

Regularization

A subset of methods used to encourage generalization in learned models, often by increasing the difficulty for a model to learn the fine-grained details of training data.

So, the next question might be, how do we know when to stop? In truth, the only real way to know is to run the model on data that isn't in your training dataset. This is typically done using a *second* test dataset called a "validation set". In some circumstances, if we used our test set for knowing when to stop, we could actually *overfit to our test set*. So, as a general rule, we don't use it to control training. We use a validation set instead.

You can see an example of validation being used in the github notebook "Chapter 6: Early Stopping", causing our previous net to stop at iteration 20.

Industry Standard Regularization: Dropout

The Method: randomly turning neurons off (setting to 0) during training.

So, this regularization technique really is as simple as it sounds. During training, you randomly set neurons in your network to zero (and usually the deltas on the same nodes during backpropagation, but you technically don't have to). This causes the neural network to train exclusively using *random subsections* of the neural network. Believe it or not, this regularization technique is generally accepted as the go-to, state-of-the-art regularization technique for the vast majority of networks. It's methodology is simple and inexpensive, although the intuitions behind *why* it works are a bit more complex.

Why does Dropout Work? (perhaps oversimplified)

Dropout makes our big network act like a little one by randomly training little subsections of the network at a time, and little networks don't overfit.

It turns out that the smaller a neural network is, the less it is able to overfit. Why? Well, small neural networks don't have very much expressive power. They can't latch onto the more granular details (i.e. noise) that tend to be the source of overfitting. They only have "room" to capture the big, obvious, high level features.

This notion of "room" or "capacity" is actually a really important one for you to keep in your mind. You can think of it like this. Remember our "clay" analogy from a few pages ago? Imagine if your clay was actually made of "sticky rocks" that were the size of dimes. Would that clay be able to make a very good imprint of a fork? Of course not! Why? Well, those stones are much like our weights. They form around our data, capturing the patterns we're interested in. If we only have a few, larger stones, then it can't capture nuanced detail. Each stone instead is pushed on by large parts of the fork, more or less *averaging* the shape (ignoring fine creases and corners).

Imagine again clay made up of very fine-grained sand. It's actually made up of millions and millions of small stones that can fit into every nook and cranny of a fork. This is what gives *big* neural networks the expressive power they often use to overfit to a dataset.

So, how do we have the power of a large neural network with the resistance to overfitting of the small neural network? We take our big neural network and turn off nodes randomly. What happens when you take a big neural network and only use a small part of it? Well, it behaves like a small neural network! However, when we do this randomly over potentailly millions of different "sub-networks", the sum total of the entire network still maintains its expressive power! Neat, eh?

Why Dropout Works: Ensembling Works

Dropout is actually a form of training a bunch of networks and averaging them

Something to keep in mind: neural networks always start out randomly. Why does this matter? Well, since neural networks learn by trial and error, this ultimately means that every neural network learns just a little bit *differently*. It might learn equally effectively, but no two neural networks are ever exactly the same (unless they start out exactly the same for some random or intentional reason).

This has an interesting property. When you overfit two neural networks, no two neural networks overfit in exactly the same way. Why? Well, overfitting only occurs until every training image can be predicted perfectly, at which point the error == 0 and the network stops learning (even if you keep iterating). However, since each neural network starts by predicting randomly, then adjusting its weights to make better predictions, each network inevitably makes different mistakes, resulting in different updates. This culminates in a core concept.

While it is very likely for large, unregularized neural networks to overfit to noise, it is very *unlikely* for them to overfit to the **same** noise.

Why do they not overfit to the same noise? Well, they start randomly, and they stop training once they have learned enough noise to disambiguate between all the images in the training set. Truth be told, our MNIST network only needs to find a handful of random pixels that happen to correlate with our output labels to overfit. However, this is contrasted with, perhaps, an even more important concept.

Neural networks, even tough they are randomly generated, still start by learning the biggest, most broad sweeping features before learning much about the noise.

The takeaway is this; if you train 100 neural networks (all initialized randomly), they will each tend to latch onto different noise but similar broad *signal*. Thus, when they make mistakes, they often make *differing* mistakes. This means that if we allowed them to vote equally, their noise would tend to cancel out, revealing only what they have all learned in common, *the signal*.

Dropout In Code

Here's how you actually use Dropout in the real world

In our MNIST classification model, we're going to add Dropout to our hidden layer, such that 50% of the nodes are turned off (randomly) during training. Perhaps you will be surprised that this is actually only a 3 line change in our code. Below you can see a familiar snippet from our previous neural network logic with our dropout mask added.

```
layer_0 = images[i:i+1]
dropout_mask = np.random.randint(2,size=layer_1.shape)

layer_1 *= dropout_mask * 2
layer_2 = np.dot(layer_1, weights_1_2)

error += np.sum((labels[i:i+1] - layer_2) ** 2)

correct_cnt += int(np.argmax(layer_2) == \
               np.argmax(labels[i+i+1]))

layer_2_delta = (labels[i:i+1] - layer_2)
layer_1_delta = layer_2_delta.dot(weights_1_2.T)\
               * relu2deriv(layer_1)

layer_1_delta *= dropout_mask

weights_1_2 += alpha * layer_1.T.dot(layer_2_delta)
weights_0_1 += alpha * layer_0.T.dot(layer_1_delta)
```

Indeed, to implement Dropout on a layer (in this case layer_1), one simply multiplies the layer_1 values by a random matrix of 1s and 0s. This has the affect of randomly "turning off" nodes in layer_1 by setting them to equal 0. Note that our dropout_mask uses what's called a "50% bernoulli distribution" such that 50% of the time, each value in the dropot_mask is a 1, and (1 - 50% = 50%) of the time, it is a 0.

This is followed by something that may seem a bit peculiar. We multiply layer_1 by two! Why do we do this? Well, remember that layer_2 is going to perform a *weighted sum* of layer_1. Even though it's weighted, it's still a **sum** over the values of layer_1. Thus, if we turn off half the nodes in layer_1, then that sum is going to be cut in half! Thus, layer_2 would increase its sensitivity to layer_2, kindof like a person leaning closer to a radio when the volume is too low to better hear it. However, at test time, when we no longer use Dropout, the volume would be back up to normal! You may be surprised to find that this throws off layer_2's ability to *listen* to layer_1. Thus, we need to counter this by multiplying layer_1 by (1 / the percentage of turned on nodes). In this case, that's 1/0.5 which equals 2. In this way, the volume of layer_1 is the same between training and testing, despite Dropout.

```
import numpy, sys
np.random.seed(1)
def relu(x):
    return (x >= 0) * x # returns x if x > 0
                        # returns 0 otherwise

def relu2deriv(output):
    return output >= 0 #returns 1 for input > 0

alpha, iterations, hidden_size = (0.005, 300, 100)
pixels_per_image, num_labels = (784, 10)

weights_0_1 = 0.2*np.random.random((pixels_per_image,hidden_size)) - 0.1
weights_1_2 = 0.2*np.random.random((hidden_size,num_labels)) - 0.1

for j in range(iterations):
    error, correct_cnt = (0.0,0)
    for i in range(len(images)):
        layer_0 = images[i:i+1]
        layer_1 = relu(np.dot(layer_0,weights_0_1))
        dropout_mask = np.random.randint(2, size=layer_1.shape)
        layer_1 *= dropout_mask * 2
        layer_2 = np.dot(layer_1,weights_1_2)

        error += np.sum((labels[i:i+1] - layer_2) ** 2)
        correct_cnt += int(np.argmax(layer_2) == \
                                        np.argmax(labels[i:i+1]))
        layer_2_delta = (labels[i:i+1] - layer_2)
        layer_1_delta=layer_2_delta.dot(weights_1_2.T)*relu2deriv(layer_1)
        layer_1_delta *= dropout_mask

        weights_1_2 += alpha * layer_1.T.dot(layer_2_delta)
        weights_0_1 += alpha * layer_0.T.dot(layer_1_delta)

    if(j%10 == 0):
        test_error = 0.0
        test_correct_cnt = 0

        for i in range(len(test_images)):
             layer_0 = test_images[i:i+1]
             layer_1 = relu(np.dot(layer_0,weights_0_1))
             layer_2 = np.dot(layer_1, weights_1_2)

             test_error += np.sum((test_labels[i:i+1] - layer_2) ** 2)
             test_correct_cnt += int(np.argmax(layer_2) == \
                                        np.argmax(test_labels[i:i+1]))

        sys.stdout.write("\n" + \
             "I:" + str(j) + \
             " Test-Err:" + str(test_error/ float(len(test_images)))[0:5] +\
             " Test-Acc:" + str(test_correct_cnt/ float(len(test_images)))+\
             " Train-Err:" + str(error/ float(len(images)))[0:5] +\
             " Train-Acc:" + str(correct_cnt/ float(len(images))))
```

Dropout Evaluated on MNIST

Here's how you actually use Dropout in the real world.

If you remember from before, our neural network (without Dropout) previously reached a test accuracy of 75.9% before falling down to finish training at 62.3% accuracy. When we add dropout, our neural network instead behaves this way.

```
I:0 Test-Err:0.681 Test-Acc:0.57 Train-Err:0.926 Train-Acc:0.388
I:10 Test-Err:0.495 Test-Acc:0.746 Train-Err:0.474 Train-Acc:0.769
I:20 Test-Err:0.454 Test-Acc:0.77 Train-Err:0.429 Train-Acc:0.824
I:30 Test-Err:0.460 Test-Acc:0.762 Train-Err:0.402 Train-Acc:0.835
I:40 Test-Err:0.467 Test-Acc:0.775 Train-Err:0.392 Train-Acc:0.836
I:50 Test-Err:0.460 Test-Acc:0.773 Train-Err:0.381 Train-Acc:0.843
I:60 Test-Err:0.458 Test-Acc:0.777 Train-Err:0.362 Train-Acc:0.869
I:70 Test-Err:0.466 Test-Acc:0.758 Train-Err:0.356 Train-Acc:0.879
I:80 Test-Err:0.454 Test-Acc:0.764 Train-Err:0.348 Train-Acc:0.879
I:90 Test-Err:0.454 Test-Acc:0.758 Train-Err:0.332 Train-Acc:0.895
I:100 Test-Err:0.452 Test-Acc:0.754 Train-Err:0.311 Train-Acc:0.905
I:110 Test-Err:0.456 Test-Acc:0.754 Train-Err:0.316 Train-Acc:0.903
I:120 Test-Err:0.450 Test-Acc:0.753 Train-Err:0.312 Train-Acc:0.903
I:130 Test-Err:0.450 Test-Acc:0.772 Train-Err:0.292 Train-Acc:0.913
I:140 Test-Err:0.455 Test-Acc:0.776 Train-Err:0.306 Train-Acc:0.918
I:150 Test-Err:0.456 Test-Acc:0.772 Train-Err:0.289 Train-Acc:0.927
I:160 Test-Err:0.464 Test-Acc:0.765 Train-Err:0.295 Train-Acc:0.917
I:170 Test-Err:0.462 Test-Acc:0.762 Train-Err:0.284 Train-Acc:0.928
I:180 Test-Err:0.466 Test-Acc:0.764 Train-Err:0.282 Train-Acc:0.922
I:190 Test-Err:0.468 Test-Acc:0.766 Train-Err:0.274 Train-Acc:0.934
I:200 Test-Err:0.472 Test-Acc:0.762 Train-Err:0.283 Train-Acc:0.94
I:210 Test-Err:0.478 Test-Acc:0.753 Train-Err:0.277 Train-Acc:0.93
I:220 Test-Err:0.471 Test-Acc:0.77 Train-Err:0.263 Train-Acc:0.938
I:230 Test-Err:0.474 Test-Acc:0.772 Train-Err:0.260 Train-Acc:0.937
I:240 Test-Err:0.488 Test-Acc:0.759 Train-Err:0.272 Train-Acc:0.935
I:250 Test-Err:0.480 Test-Acc:0.778 Train-Err:0.258 Train-Acc:0.939
I:260 Test-Err:0.488 Test-Acc:0.768 Train-Err:0.262 Train-Acc:0.943
I:270 Test-Err:0.487 Test-Acc:0.769 Train-Err:0.254 Train-Acc:0.945
I:280 Test-Err:0.503 Test-Acc:0.772 Train-Err:0.262 Train-Acc:0.938
I:290 Test-Err:0.509 Test-Acc:0.77 Train-Err:0.247 Train-Acc:0.949
```

Not only does the network instead peak out at a score of 77.8%, it also doesn't over fit nearly as badly, finishing training with a testing accuracy of 77%. Notice that the dropout also slows down the Training-Acc, which previously went straight to 100% and then stayed there.

This should point to what Dropout really is. It's noise. It makes it more difficult for the network to train on the training data. It's like running a marathon with weights on your legs. It's harder to train, but when you take them off for the big race, you end up running quite a bit faster becasue you trained for something that was much more difficult.

Batch Gradient Descent

A method for increasing the speed of training and the rate of convergence.

In the context of this chapter, I would like to briefly apply a concept introduced several chapters ago, the concept of mini-batched stochastic gradient descent. I won't go into too much detail, as it's something that's largely taken for granted in neural newtork training. Furthemore, it's a very simple concept that doesn't really get more advanced even with the most state of the art neural networks. Simply stated, previously we trained one training example at a time, updating the weights after each example. Now, we're going to train 100 training examples at a time averaging the weight updates between all 100 examples. The code for this training logic is on the next page, and the training/testing output is below.

```
I:0 Test-Err:0.849 Test-Acc:0.325 Train-Err:1.347 Train-Acc:0.159
I:10 Test-Err:0.603 Test-Acc:0.643 Train-Err:0.602 Train-Acc:0.659
I:20 Test-Err:0.547 Test-Acc:0.708 Train-Err:0.538 Train-Acc:0.726
I:30 Test-Err:0.522 Test-Acc:0.735 Train-Err:0.503 Train-Acc:0.745
I:40 Test-Err:0.502 Test-Acc:0.746 Train-Err:0.487 Train-Acc:0.746
I:50 Test-Err:0.491 Test-Acc:0.752 Train-Err:0.466 Train-Acc:0.791
I:60 Test-Err:0.485 Test-Acc:0.762 Train-Err:0.447 Train-Acc:0.789
I:70 Test-Err:0.474 Test-Acc:0.767 Train-Err:0.439 Train-Acc:0.791
I:80 Test-Err:0.473 Test-Acc:0.769 Train-Err:0.441 Train-Acc:0.794
I:90 Test-Err:0.470 Test-Acc:0.759 Train-Err:0.439 Train-Acc:0.804
I:100 Test-Err:0.470 Test-Acc:0.756 Train-Err:0.419 Train-Acc:0.815
I:110 Test-Err:0.465 Test-Acc:0.758 Train-Err:0.413 Train-Acc:0.828
I:120 Test-Err:0.467 Test-Acc:0.769 Train-Err:0.407 Train-Acc:0.814
I:130 Test-Err:0.465 Test-Acc:0.773 Train-Err:0.403 Train-Acc:0.818
I:140 Test-Err:0.469 Test-Acc:0.768 Train-Err:0.413 Train-Acc:0.835
I:150 Test-Err:0.469 Test-Acc:0.771 Train-Err:0.397 Train-Acc:0.845
I:160 Test-Err:0.469 Test-Acc:0.776 Train-Err:0.403 Train-Acc:0.84
I:170 Test-Err:0.471 Test-Acc:0.772 Train-Err:0.394 Train-Acc:0.859
I:180 Test-Err:0.468 Test-Acc:0.775 Train-Err:0.391 Train-Acc:0.84
I:190 Test-Err:0.466 Test-Acc:0.78 Train-Err:0.374 Train-Acc:0.859
I:200 Test-Err:0.469 Test-Acc:0.783 Train-Err:0.392 Train-Acc:0.862
I:210 Test-Err:0.470 Test-Acc:0.769 Train-Err:0.378 Train-Acc:0.861
I:220 Test-Err:0.466 Test-Acc:0.782 Train-Err:0.369 Train-Acc:0.864
I:230 Test-Err:0.465 Test-Acc:0.786 Train-Err:0.368 Train-Acc:0.87
I:240 Test-Err:0.469 Test-Acc:0.782 Train-Err:0.369 Train-Acc:0.869
I:250 Test-Err:0.470 Test-Acc:0.776 Train-Err:0.362 Train-Acc:0.876
I:260 Test-Err:0.466 Test-Acc:0.787 Train-Err:0.371 Train-Acc:0.859
I:270 Test-Err:0.469 Test-Acc:0.778 Train-Err:0.357 Train-Acc:0.877
I:280 Test-Err:0.469 Test-Acc:0.778 Train-Err:0.371 Train-Acc:0.878
I:290 Test-Err:0.466 Test-Acc:0.775 Train-Err:0.359 Train-Acc:0.873
```

Notice that our training accuracy has a bit of a smoother trend to it than it did before. Furthermore, we actually reach a slightly higher testing accuracy of 78.7%! Taking an average weight update consistently creates this kind of phenomenon during training. As it turns out, individual training examples are very noisy in terms of the weight updates they generate. Thus, averaging them makes for a smoother learning process.

```
import numpy as np
np.random.seed(1)

def relu(x):
    return (x >= 0) * x # returns x if x > 0

def relu2deriv(output):
    return output >= 0 # returns 1 for input > 0


alpha, iterations = (0.1, 300)
pixels_per_image, num_labels, hidden_size = (784, 10, 100)

weights_0_1 = 0.2*np.random.random((pixels_per_image,hidden_size)) - 0.1
weights_1_2 = 0.2*np.random.random((hidden_size,num_labels)) - 0.1

for j in range(iterations):
    error, correct_cnt = (0.0, 0)
    for i in xrange(len(images) / batch_size):
        batch_start, batch_end = ((i * batch_size),((i+1)*batch_size))

        layer_0 = images[batch_start:batch_end]
        layer_1 = relu(np.dot(layer_0,weights_0_1))
        dropout_mask = np.random.randint(2,size=layer_1.shape)
        layer_1 *= dropout_mask * 2
        layer_2 = np.dot(layer_1,weights_1_2)

        error += np.sum((labels[batch_start:batch_end] - layer_2) ** 2)
        for k in xrange(batch_size):
            correct_cnt += int(np.argmax(layer_2[k:k+1]) == \
                            np.argmax(labels[batch_start+k:batch_start+k+1]))

        layer_2_delta = (labels[batch_start:batch_end]-layer_2)/batch_size
        layer_1_delta = layer_2_delta.dot(weights_1_2.T)\
             * relu2deriv(layer_1)
        layer_1_delta *= dropout_mask

        weights_1_2 += alpha * layer_1.T.dot(layer_2_delta)
        weights_0_1 += alpha * layer_0.T.dot(layer_1_delta)
```

The first thing you'll notice when running this code is that it runs way faster. This is because each "np.dot" function is now performing 100 vector dot products at a time. As it turns out, CPU architectures are way faster at performing dot products batched in this way.

There's actually more going on here, however. Notice that our alpha is 20x larger than it ways before. We can increase this for a rather fascinating reason. Consider if you were trying to find a city using a very wobbly compass. If you just looked down, got a heading, and then ran 2 miles, you'd likely be way off course! However, if you looked down, took 100 headings and then averaged them, running 2 miles would probably take you in the general right direction. Thus, because we're taking an average of a noisy signal (i.e., the average weight change over 100 training examples), we can take bigger steps! You will generally see batching ranging from size 8 to as high as 256. Generally, researchers pick numbers randomly until they find a batch_size/alpha pair that seems to work well.

Conclusion

In this chapter we have addressed two of the most widely used methods for increasing the accuracy and training speed of almost any neural architecture. In the following chapters, we will pivot from sets of tools that are universally applicable to nearly all neural networks to special purpose architectures that are advantageous for modeling specific types of phenomeon in data. See you there!