In this set of videos, we will introduce

the basic concepts behind working with

neural networks. Now, neural networks and

deep learning are behind most of the artificial

intelligence that shapes our every

day life today. Think of all the cool

features in our phones, ranging from face-recognition

to autocorrect to texts autocompletes,

voicemail detects previews. Also the way that we

find what we need on the Internet using predictive

internet searches, content or product

recommendations, and even self-driving cars. Also, many of the classification

and regression problems that you need to solve at your business are

going to end up being good candidates for

neural networks and deep learning as well. Now there are several

Watson applications and artificial intelligence

APIs that help you infuse artificial intelligence

into your business. Here we have some of the

most used with links to live demos that you can

explore here on your own. As you go through some of these, think about ways you can use these applications

within your business. Whether it's identifying

the pieces of an image, coming up with an efficient

translation into a foreign language,summarizing and

classifying comments or reviews of your product, results finding whether those

comments have positive, negative, or perhaps

a neutral tone. Now it's often noted that the biology of the

brain serves as an inspiration for the

mathematical models that make up our neural networks. The idea being that

the brain functions by firing neurons along a chain, where one neuron

gets signals from prior neurons and according to the firings of prior neurons, the next neurons decide

where to generate signals or not generate signals

according to those inputs. Those signals that were

activated then pass on signals down that chain

to the next neurons. By layering many

neurons together, we end up creating a

very complex model. Now moving to the

actual neural networks, we can think of it as a

complicated computation engine. We're going to train it

using our training data. Train our neural net model. Then we'll use that

trained neural net model to generate predictions

at using new data. Note here that similar

train-test approach as we did with

supervised learning, which will become of

utmost importance as we create our neural networks. Now that closes out this video. In the next video, we're going to dive

into a single one of these cells to see how data flows in and how data

flows out from each one from layer to layer.

I'll see you there.

Now let's zoom in on a single node in the middle

of our basic neural network. First that node will

get input values from the previous layer

wherever that node lies. Those input values

are then going to be combined via weights from each

of those different values, similar to basic multiple

linear regression. Then that combination

of weights is going to be transformed

similar to how logistic regression transforms

a linear combination to squash those values

between 0 and 1, and that transform value is used as input

for the next layer. Now let's add in some variables to paint this

process a bit clear. We can have as our three

input values: X1, X2, and X3, and we'll assume also an intercept term with

that value equal to 1, as we do with

multiple regression. We also have the respective

weights for each one of our different

values: W1, W2, and W3, as well as b, and our model is

going to learn each one of these weights

as well as the b. As mentioned, we will multiply each value by its weight as

we do with linear regression, and end up with some

output value z. Finally we're going to use an activation

function, like I said, similar to logistic regression or even logistic regression, to transform that output and use that value as input

for the next layer. Now without this activation

function we are restricted to only linear output or a linear combinations

of our inputs, and no matter how many

layers deep we go, we're still just working with a linear combination

of our features. It's going to be this

activation function that allows for the great

flexibility with respect to how we consider the model outputs given our model inputs using

a neural network. Now, some notation that'll

be worth getting familiar with as we walk through

working with neural networks. We have z, which is going

to be the net input or the linear combination of the inputs prior

to activation. Essentially the output of

just that linear regression. We'll have our bias term or

that b that we just saw, which is also similar

to our bias term within linear regression, we'll have f, our activation function, that non-linear function we use to

transform the output of z. Then we have a, our output layer, or the value once we take f of z, once we transform z, that we ultimately pass

through to our next layer. Now with this syntax in mind, as well as that basic unit

that we just walked through, that basic neuron, we've

seen that there is a lot of relation between that neuron

and the logistic regression. When we choose f of z equals to 1 over 1 plus e

to the negative z, where z is our output of just the linear part

of that neuron, we were actually

looking at something very similar to

logistic regression, and what we have here with z, z is just going to be equal

to that intercept term, plus the sum of each one

of the different inputs, multiplied by their

respective weights, which we've expanded out here. Our neuron is then

simply just a unit of logistic regression where we

have the different weights that we learn are just

the co-efficients for logistic regression, the inputs are the different

variables that we have here, and the bias term is

that constant term. It all relates back to our

basic logistic regression. Because logistic regression and our neural network in a way can accomplish the same task if we're trying to

accomplish classification, we want to ensure

that when we move to neural network that we actually

need a more complex mode, that we don't just

need this single unit, but we need multiple units

and perhaps multiple layers, and that's when we would switch

over to neural networks. The trade-off being that you

may be able to come up with a more complex boundary

with neural networks, but you'll lose a lot of the explanatory value that you have with logistic regression. What we have here is going to be a sigmoid function which we

use for logistic regression, as well as our activation

here when we talked about the neuron and the output

for the neural network. What our sigmoid

function will do, will take that linear combination and create a non-linear function. As we see here, and we

have non-linearity, not a straight line here, and squash those values

between 0 and 1, which will be useful as we walk through the different steps

of our neural network. Now there are nice properties of the sigmoid

function which make it particularly useful when passing it through our neural network. Let's say we set our

activation function to the sigmoid function, what's special

about it is we will want to take the derivative

of it at each step, and we'll see why later

on down this course, but in order to take this derivative we'd use

the quotient rule, where we take the derivative

of the top multiplied by the bottom minus the top multiplied by the

derivative of the bottom, all over that bottom squared, and that's going to

give us our derivative. Now, if we're trying to take the derivative of the

sigmoid function, plugging in each one of

the different values we see that the derivative

of 1 should be 0. That first value will be 0. Then we multiply that top value, which is just 1, by the

derivative of the bottom, which is just going to be

negative e to the negative z, and then that's all over 1 plus e to the negative z squared. Then we end up with

e to the negative z over 1 plus e to the

negative z squared, which we can expand out, adding and subtracting

1 to that top value, and that should

make clear that we have 1 plus e to the negative z, minus 1 all over those same values of 1 plus

e to the negative z squared. We can then cross out that top and bottom values

as we see here, and we end up with

rather than squared, 1 over 1 plus e to

the negative z, minus 1 over 1 plus e to

the negative z squared. Factor that out. We just have factoring out that 1 over 1 plus e to

the negative z. We have that multiplied by 1 minus 1 plus e to

the negative of z, which is just our

original function multiplied by 1 minus our

original sigmoid function. Nice step if we want to actually find the

derivative along the way, which will be useful as we try to compute our

neural network. Now that closes out this video. In the next video we'll introduce a means of actually using

this within Python, after focusing in on a single

perception and then seeing how each layer will

connect to one to the other. I'll see you there.

So let's zoom back into the single node. One way of working with

just a single neuron. What we have here is a perception and this is the basis upon which all

of Neural Networks are built. Now note here that we have as

before our input values x1, x2 and x3 as well as our intercepts and then our weights and

our beta that we're going to learn. And in this example, we're going to be

using logistic regression using that sigmoid activation function

that we just discussed. Now, if we were to change and

look at actual values just to make clear how this actually looks in practice,

we can have the values as inputs. Imagine that we have a row

with feature 1 equal to .9, feature 2 equal to .2,

feature 3 equal to .3 and then our w1w2 and w3 are 2,3 and

-1 with a b of .5. We can then calculate the actual z value. That would be input once we have

each one of these values into our activation function. That activation function is 1/(1 + e2

whatever rz that we calculated was. And we'd end up with a value of .93. And that would be the output

of this particular node. So our note output is .93. So why not just use a single neuron? Why do we need to have a larger network

where you have one stack on top of the other? If we had just a single neuron as we

would if we were just doing logistic regression now to only permit

a linear decision boundary. When we move onto stacking one

layer on top of the other. Were able to come up with a much

more complex decision boundary and most of our real world problems will

probably be much more complicated than just that linear decision

boundary that we can learn with something like logistic regression or

something with just one unit. So in order to take our inputs and

pass them through an get our different outputs as we see here, we'd be

working with a multi-layer perceptron. So we saw that one unit perceptron,

we add on each one. And we see here that we have

this feed-forward structure. Where we have our inputs of x1, x2 and x3. Those will each be inputs

into the next layer. If you look at each one of the arrows

X1 goes to each one of the different perceptions on that next layer. As does X2 and as does X3. And then not next layer, the second layer is connected to

every value in the third layer. And so on and so forth until we

get our output of y1, y2 and y3. Now, in order to create the multi-layer

perception and practice using Python, we're going to go over the sklearn

version of creating this Neural Network. Now, something to note is that we

can make this simple multi-layer perception using sklearn. But as we move on to more complex models,

you will see that we're going to move away from Scikit-Learn and

start working with a library called Karas. But for now,

let's continue to focus on sklearn. So as usual, we're going to import

from sklearn.neural\_network were going to import the MLPClassifier. We then need to specify

are activation function. So we pass in the different arguments

while we initiate a class of this MLPClassifier. And some of the arguments that you

see here are the hidden layer sizes. So this will actually be the sizes of each

layer between your input and your output. So as we saw before we input x1,

x2, x3 and then we have certain

amounts of hidden layers. And we're saying here the size of

each one of those hidden layers. So the fact that this

toopols only a size 2. That means that there's 2 hidden layers,

one of size 5, one of size 2. If we wanted three and we wanted the third

one to be of size 5, we can do 5,2,5. So that's how the hidden layer

sizes argument will work. And then the activation

function that we want to use. We've seen so far that we've

only used the sigmoid function. By default sklearn will

actually use the relu-function, which will learn a bit later. But because we want to stay in line

with what we've discussed so far, we're going to set the activation

equal to logistic here. And logistic is just the same

as setting equal to sigmoid. We can then, as usual, fit and

predicts given our data. So we pass into our fit our x\_train and

our y\_train, and then we can pass into our mlp.predict

our holdout set our x\_test and see how well we performed

on this holdout set. Now let's closeout this video and

then the next video we're going to go into some of the common terminology used

for the multilayer perception as well as some intuition behind the basic

math that brings us all together. All right, I'll see you there.

Now let's walk through some of the

important terminology that we should keep in mind when working with a neural

network and here a multilayer perceptron, as well as some of the basics as to how

we get from this first layer up until the final layer that we have from

the X's is up until the y's. So first off, we have our different

weights and those weights will determine how do we combine each one of the

different layers along our neural network. So each one of these arrows that

will connect X1 to each point, each node within that next layer, as well as all the lines between

the second layer and the third layer. These will all signify

be specific weights and how to combine each one of

these different layers. We have our input layer which is

just going to be our input data set. Here just to make it especially clear,

we can imagine that this first X1, X2, X3 is just going to be the first row, where X1 is feature 1,

X2 is featured 2 and X3 is feature 3. We then have our hidden layers and those

are going to be all of these purple nodes that fall between our input layer and what we will define right

now as our output layer. So everything between our input layer

on output layer are going to be called are hidden layers. And those hidden layers as we

specified as we walked through the Python syntax can be defined

however many layers we'd like. We can say we want five hidden layers and

they would be five different columns of notes in between

our input layer and our output layer. That something that we would predefine

an all the weights would connect to each of those in order to learn

this complex model feeding from our input layer through the hidden

layers out through the output layer, which will be our actual predictions. The weights that we said over

the different arrows are going to be represented by matrices. And each of those different

matrices will again just be the way that we combine

each layer step by step. And those matrices will have to be of

the appropriate shape to ensure that if we have an input that's

going to be three vector, that it transforms it into

a four vector in the next layer. And then maintains that four vector in the

next layer, and then brings that down to a three vector in that final layer, and

I'll walk through this in just a second. Our net input will be the sum of

the weighted inputs, so that's going to be your Z values, and that's going to be

again similar to your linear regression. So X1 times some weight plus X2 times

some weight plus X3 times some weight or eq will equal one of your values of Z. And then we will have four different

values for that first layer. So our Z is actually going

to be a four vector as well, our Z2, and

then our Z3 will be at three vector. We then finally have our activation

values and those activation values are just going to be taking those Z

values that we just discussed and passing them through our

activation function. So I'm going to briefly skip over a0 here,

but a1 should be a four vector as well, or we just take that Z1 and pass it

through for example, each one of those different values in that four vector,

pass it through the sigmoid function. We can do the same for

a2 passing through Z2. And then for a3 we can pass Z3,

probably here through softmax layer in order to give the predicted

probabilities that we'd want output for this classification

problem that we have here. Now if we go back to a0,

a0 is signifying that we want any a to be pass into the next layer. Even though we're not doing

anything to the X1, X2, and X3, that is going to be fed as

input into the next layer, so will often call that a0 just for

simplicity's purposes. So imagining working with just

a single row of data again. So if we have a single data

point with a certain amount, so that's a single row with

a certain amount of features. Our W1 or first weight would be a 3 by 4

matrix, taking the input values of X1, X2 and X3,

which would be a one by three matrix and multiplying that one by three

matrix by the 3 by 4 matrix W 1. And that will result in our 1 by

4 matrix which will be our Z1. We could then pass all the values of

Z1 through our activation function. And that would result in

a1 another four vector. In order to make this as clear as

possible, and as we saw in the prior side, we can think of our input values as a0. And every a, a0, a1, a2 will be the value

that's passes input into the next layer. And again our Z1 is equal to

the dot product of x and W 1 and our a1 is just going to be

the activation function of Z1 and that will be passed on to the next layer. Now to take this a step further to see how

this computes through the entire process of our neural network for a single row. We are planning to start with a vector

representing that row in our case, that was a row vector of length 3. And plan to end with an output of

a row vector of length 3 as well, which means in this example we're probably

performing classification with output of three classes. Now we showed how we got Z1 as

a dot product of x and W 1. That allows us to calculate a1 by

taking the aggravation function of Z1. Z2 or the second layer of Z is

calculated as a linear combination of each of those a1's

that we just calculated. And in order to get a linear combination

of the correct dimensions, we have W2 which matches up the shape of a1 and our

eventual shape of the values we want a2. A2 is then again just

the activation function of Z2. And then Z3 is then again the linear

combination of that prior output of a2. So we need a new weight matrix W3. And once we get the linear combination

of a2 and since this is the final layer, we just take the softmax of Z3 in order to

give us the predicted probabilities for each one of our different classes and

that will be our predicted, y. Now, in practice were not just working

with a single row at a time, but rather we'd be working with

an entire data set worth of rows. But in order to calculate this generalized

version of our multilayer perception, our equations should look very similar,

or exactly the same? So this time we're in

putting an n by 3 matrix, where n is the number of rows still

working with three columns though. And our output should also be an n by 3

matrix with a predicted probability for each one of our different rows. Now the math should be the same though,

as we saw before, but this time the dot product of x and W is

now just going to be an n by 4 matrix or whatever the size of our next layer is. Rather than it just being

1 by 4 we're now n by 4. So if you imagine again that that X is

an n by 3, we can have our be 3 by 4 so that we'd end up with

an end by 4 matrix for Z1. We can then take the activation function

for all of the outputs that we get for Z1 and end up with our output

from that first layer. And again we have the appropriate matrix

to get the linear combination of each one of those outputs for each one of

the different rows and end up with Z2. We pass each of those Z2 through

the activation function to get a2. That a2 will be the output for the second

layer, and input into the third layer and that will give us Z3 when we take

the linear combination of a2 and W 3. And taking that Z3 now for multiple rows, we can take the Softmax and

end up with predicted probabilities for each one of the three classes for

all of our n rose. So that expands it out to the amount

of rose within our entire data set. Now there are many deep learning

approaches which we're going to discuss throughout this course. And along with these basic groupings,

there's also much more being developed. So quick overview,

we have the neural network models, which are just going to be your multilayer

perceptron and feed forward networks. And this is going to be applied to many

traditional predicted problems such as just classification and

regression that we've discussed. So, We have recurrent neural networks,

and we have here the classes of RNN and LSTM, long short term memory,

RNN is recurrent neural network. And this is going to be useful for

modeling sequences. So this will be useful for

time series, where, maybe each one of the different steps along the way

are dependent on prior steps. Or sentence prediction, where each one of the different words

may be dependent on prior words. We'll have convolutional neural networks

or CNN, and that's going to be very useful for a feature and object recognition

in visual data, as it will take all of the surrounding features and take them

in as context moving forward as well. As well as being used at times for forecasting as well,

where I can take points on either end or see some type of patterns within the data,

in order to predict future values. And then it can also be used with

unsupervised pre-trained networks with Autoencoders, Deep Belief Networks,

and Generative Adversarial Networks. And there's going to be many uses

including, generating actual images, labeling some outcomes,

as well as dimensionality reduction, using deep learning. And we'll discuss many of

these throughout this course. Now that closes our introduction

to neural networks. In the next video,

we will begin to discuss the optimization, that's needed in order to come up with

our weights, using gradient descent, which will be a key factor in learning

each one of our neural network models. Alright, I'll see you there.

Gradient Descent

In this section, we will

discuss gradient descent, one of the most crucial

components that make learning the parameters of our neural

nets actually possible. The learning goals for

this session are going to include going over

basic gradient descent, as well as how to do stochastic gradient descent or update our parameters by going through one

observation at a time. Then finally mini-batch

gradient descent, which will update

our parameters using just a certain subset amount of our observations

within our data. For gradient descent,

we're going to start with a cost

function J of Beta, which looks like the following. Our goal is to find the Beta values at which this

cost function is minimized. Now in order to find that Beta which we

see as our x-axis, we initialize at some random

point on our cost function, and then we use the gradient to gradually descend towards

that minimum value. That minimum value means we

minimize the cost function, and that will be our

optimal value for the Beta. Now, let's discuss an example with linear regression to

help make this a bit clear. Imagine that you're working

with a simple regression or trying to learn two Beta

values, Beta\_naught and Beta\_1. Now in three dimensions, we have on one axis all the possible values

for Beta\_naught. On the other we have all of the possible values for Beta\_1, and on that final axis, that vertical axis,

we have the output of the cost function we're

trying to minimize for all values of

Beta\_naught and Beta\_1. Now that we have increased

the number of dimensions, we now have a more

complicated surface on which to find

that minimum value. How do we find that minimum

value if we can't tell exactly how this cost function will look for our given model? The key again is to

start at a random point. We then compute the gradients of this point in respect to

Beta\_naught and Beta\_1, and that gradient will always point in direction of

the largest increase. Now we take the negative

of that gradient and now we are pointing in the direction

of the largest decrease. Now that gradients that

we are discussing, will be a vector in that same dimensional

space as our parameters, consisting of the

partial derivatives of each one of those parameters. We have the gradients and

that's going to tell us the direction of descent for each one of our

individual parameters. As we see here, we have for every

single parameter, each one of their

partial derivatives. We can then use the gradients

and the given cost function to calculate a new point from that original initialized point. We started off with w\_naught. Now w\_1 will equal w\_naught

minus a learning rate, which we'll discuss

in just a second, multiplied by the gradient

of our cost function. We can see that we have now

moved closer to minimizing our cost function as we move down and we subtract

that gradients. Now that learning rate is going to be a tunable parameter, that will tell us how large

we want to make each one of our steps within

our cost function. We want to be careful with this because too large of a step, and we'll end up

overshooting our minimum, and too small of the step, and it'll take too long

to optimize our model. Now using the same concept

of subtracting the gradient, we can iterate to move

closer and closer to the minimum value

from that last step. Now w\_2 is going to

be equal to that w\_1 we just calculated minus the gradient of

the cost function. You see we move closer with w\_2. Then w\_3 will be the same thing subtracting from the

weights we got from w\_2, the gradients, and we see

we move a bit closer. Eventually we end up

with a global minimum. Now, the concept of a stochastic gradient

descent compared to that gradient descent

we just discussed, is to speed things up by using only a single data point to determine the gradient

and the cost function. Rather than summing

together all of the error values and then taking the gradients as we did with that vanilla

gradient descent. We instead calculate our weights by subtracting from w\_naught, the gradient given the

error for just one value. If you saw above we have the

sum over all the values, and now if we look above

x\_observation and y\_observation, we're doing this just

for one specific value. Then using the single point, we can again iterate through to continue to

update the weights. We can use for w\_1

and each one can be a different random

point but we keep using only a single random point. We keep updating our weights, moving down our cost

function and eventually, we end up hopefully near

some global minimum. But that path is going to be

much less directed due to the noise of working with

just a single data point. That's the idea of a

bit of randomness, and it being a stochastic

gradient descent. Now finally with mini-batch

gradient descent, we can choose some value n between one and the size

of the entire data set. Now perform an update for

every n training examples. Now rather than summing

over the entire data set or just one

single observation, we are summing over

a random subset of our original data,

seen our error, and taking the gradients

and moving down that line given the gradients

for that subset of values. Now we get the best

of both worlds. We can reduce the

memory relative to our original or vanilla

gradient descent where we used the

entire data set. But it'll be less

noisy and get to the optimal value

much smoother than working with stochastic

gradient descent. That's going to be

the idea behind it. Mini-batch gradient

descent, that n will be another parameter

that we'll be tuning as we work with our

neural network models. Now just to recap, in this section we discussed gradient descent or a full

batch gradient descent. Where we went through every

single row in our data set in order to update each

one of our gradients. We then discussed

stochastic gradient descent and how we can take steps according to the gradient on each one of the single

rows within our data set. Checking that error against every single row, then

updating accordingly. We discussed between the two how gradient descent

may take a long time, but will move more smoothly. Whereas stochastic

gradient descent will move more efficiently, but maybe a little bit bouncy

in regards to getting to that desired goal of

our optimal value. The compromise was this

mini-batch gradient descent, where we reached that

optimal value by not taking every single observation within our data set in order to

calculate the gradient. Not just taking a single value, but taking a

mini-batch taking say, 32 observations or

64 observations before creating an update

using that gradient. Now closes out our video

here on gradient descent. We'll get a clear picture in our next video where

we'll jump into a notebook on how

gradient descent actually works.

I'll see you there.

Notebook

Welcome to our Notebook

here on gradient descent. In this Notebook, we're going to have an overview of working with gradient descent in order to do solve simple

linear regression, as well as working with

stochastic gradient descent, which we defined in

lecture as just taking a single row and

seeing the error and moving according to just the

error on that single row compared to with vanilla

gradient descent where we use the entire data set. To start off, we're going to import all the

necessary libraries, here just importing NumPy,

pandas, and Matplotlib. We're then going to generate data from a known distribution, so we'll know the actual

values that we want to find when we do our

gradient descent. If we think about just working

with linear regression, in general, what we're trying

to solve for is some Y, where that Y is equal

to some Betas or some different

coefficients multiplied by our different values

in our X data set. Here we have Y equals b, which is just our intercept term, plus Theta\_1 times X\_1 plus Theta\_1 times X\_2

plus some error term. In order to generate

the data where we specify each one of

the different Thetas, we're going to have x1 and x2, each be random values

between zero and 10, where any value between

zero and 10 is equally likely to be picked since we're picking from a

uniform distribution. We're then going to

actually set the values for b, Theta\_1 and Theta\_2. So b is going to be 1.5, Theta\_1 is going to be equal to 2 and Theta\_2 is going

to be equal to 5. Then from there we can generate our y values as well

as our feature matrix, which will have our

x1 and x2 values. How do we do that? First thing we want to do is

we're going to set the random seed so

that you back home are saying the same

solutions that we have here. We're then going to say that

we want 100 observations, and we're going to pass

that through as our x1s and our x2s are going to be random values

between zero and 10. So np.random.uniform,

values between zero and 10, and we want 100

different observations between those values

of zero and 10. We set that equal to

our x1 and our x2, and then for our constant term, we're just going to call np.ones, which will just

create an array of ones for a certain shape that you will define and we

just define it as a one-dimensional array

with 100 different values. Then finally we're going

to add on that error term. If you recall up here, we also want to include the error term. This is to ensure

that it doesn't fit exactly and we'll set

that error term equal to just values from a

normal distribution with a mean of zero and a

standard deviation of 0.5, and again, we want

100 different values. We're then going to choose

our b, our Theta\_1, and our Theta\_2 to match with the values we defined above. Then y is just going to be equal to b times that constant term, which is just our one's, Theta\_1 times our x1 that we defined as random values

between zero and 10, and Theta\_2 times x2, which is again different

values between zero and 10 plus that small error value. We're then going to create

an array out of our x1, x2, and our constant term so that we have our x matrix

or our feature matrix. We run this, and then we can see what our

actual y-value is. That should be some combination of if we look at this x\_mat, we should have something

along the lines of two times this value and five

times this value plus 1.5, since 1.5 will just

be multiplied by 1. We'll have that for

each x1 and x2. In order to get the

right answer directly, we can look at the

closed form version of this model rather than using something like gradient descent. With linear regression, we can actually use matrix

algebra to get the exact solution that will find the maximum

likelihood or the least squares estimate

for our data set, and that's just going to be

this matrix algebra here. It's not too important, all that's important here

is to know that there's a closed form solution and

that for linear algebra, you do not necessarily

have to use gradient descent to find

each one of your parameters. Now, the reason why we introduce gradient descent is because, one, we're doing deep learning or even for many of

our other models, we can't find this

closed form solution, and we'll need to

use gradient descent to move towards

that optimal value, as we discussed in lecture. Here we're going to use sklearn's linear

regression model, as well as also using the actual matrix algebra that we have defined here, which we can just

pull out from NumPy. From sklearn, we're going to import our linear

regression model. We're going to call

linear regression. We don't want to

fit the intercepts since the intercepts

are already included in our feature values in our x\_mat already that

we defined earlier. So we set fit\_intercept

equal to false, and then we can fit our x\_mat and our y and then see what are different coefficients

that it comes up with. We can see that

it's very close to the values that we

wanted for b of 1.5, Theta\_1 of two, and

Theta\_2 of five. Now sklearn, the linear

regression model, will be using this closed

form matrix algebra in order to solve for it. So we should get

the same solution. One we call out this

equation just using NumPy. So that's just going

to be the inverse of the dot product of the

transpose and the value itself, and then the dot

product of that with x transpose and then the

dot product of that with y. Then when we look at the solution that that comes up with, again, it's exactly the same as

what we just saw with linear regression from sklearn. Now that closes out

this section of just getting an intro of the data

that we're working with. In the next section, in the next video, we're going to discuss actually solving this problem using gradient descent

as well as how to visualize that process.

I'll see you there.

Welcome back for

the second part of this demo on gradient descent. As we mentioned in

the last video, when we're working with other problems such as

working with neural networks, we are not going to have this analytical solution

that we just found. Instead, we're going to

have to move towards that optimal value

using gradient descent. In order to see this in practice, we're going to actually pick a learning rate as well as

a number of iterations, run the code and

plot the trajectory as we move towards

our gradient descent. If you recall in lecture, that means that

we're going to pick how big each one of our

step size are going to be, and then see step-by-step as we move closer to

the optimal value, how we're actually moving

towards each one of the different Thetas that

we're trying to predict. Then using that, we'll find some examples where the

learning rate is too high, too low, or just right. We're going to start off

with a learning rate of 1 times 10 to the

negative three, so 0.001, and we're going to say we

want 10,000 iterations, so 10,000 different steps, and we're going to initialize

with a value of 3, 3, 3. In order to actually

perform gradient descent, we're going to pass

in a learning rate, which we defined above, the number of iterations, and the Theta initial, all defined up here above. The initialization steps will be, we will set our Theta originally equal to

that Theta initial, which is at this point 3, 3, 3. We're then going to set the

Theta path at first equal to just a bunch of zeros in the shape of the number

of iterations plus one. So if we're doing

10,000 iterations, there'll be 10,001 rows, and each one will

be three columns. So it will say for each one

of our different values, which are B, Theta

1, and Theta 2, how we're moving closer

and closer to each one of those steps through each one

of the different iterations, and then we're setting

that first value, so Theta 0 for all the columns equal

to that Theta initial. Then just to start off, we're going to set the loss

vector equal to np.zeros, and we'll see the loss as

we move through each one of the steps to see if we continue

to minimize that loss, and we'll do that for

every single iteration. We're then going to do this

main gradient descent loop, which is going to be

what we discussed in lecture in regards to starting

at the initial point, finding the gradients, and

then using that gradient to move closer and closer

towards our optimal value. We're going to set our

prediction equal to the dot product of our different Thetas

times our x matrix. If we think about that as

taking our entire X matrix, and if we take the dot

product of the transpose of Theta and the transpose

of the x matrix, then all we're doing

is multiplying, in this case,

initializing with 3, 3, 3, three times the first

value for all of them, three times the second,

three times the third. Adding those

altogether and getting our first prediction for each one of our different Y values. Our loss vector, which

we defined up here, we'll then say that first loss will be equal to the sum of the square of what we predicted minus the actual values

minus what we predicted, so we'll get the mean squared error or just the squared error, and then our gradient vector, which we didn't go

through in lecture, but just to know

what the gradient actually looks like as we're taking those partial derivatives. What it'll look like is going to be that error on the prediction, so y minus y pred, take that and the dot

product of x map. So that's actually going to be equal to that gradient vector. That's how we can come up

with that gradient vector, and it will be of the

size that we need in regards to subtracting, or in this case,

this will actually be the negative of it. So we're actually later

on going to add it on. We see here that we add

on that gradient vector, but that's actually

going to be equal to the negative of

our gradient vector. We divide that by the number of observations that we have, and we'll use that

in order to move a step closer towards

our actual data. So at first, our

Theta is 3, 3, 3. We got that gradient vector, and we set 3, 3, 3 plus that learning rate multiplied by that

gradient vector, which should also be of the

same shape as that 3, 3, 3. So row vector with three values. Then we say that Theta i plus 1 is equal to that Theta

that we just found. So we reset the

Theta and then we go back through the four loop

using that new Theta, and coming up with our new

prediction, our new error, our new gradients, and

then our new Theta values. Then we're going

to return after we go through that entire four loop, the entire Theta path

as we go through each one of the

different iterations, as well as the full loss factor. How far off are we from our actual solution as

we move down the line? If we recall, that

loss vector is just going to be the

sum squared error. So we have our gradient

descent function, and then we're going to

actually plot this out. I'm going to quickly

walk through this. I'm not going to go through

every single line of code, but I do want you to

get some intuition as to what we're plotting here. We have our true coefficients, which are just equal to the B, the Theta 1, and Theta 2

that we defined earlier, and then we say plot i, j, and what we're

doing here is we're plotting two of these

different values, so either B versus Theta 1, or B versus Theta 2, or Theta 1 versus Theta 2. Those are going to be

each of our three plots. So plot i, j, we're going to plot the

actual true coefficients. The true coefficient of, if we say i and j are

equal to v and Theta 1, then we'll say we

want the 0th value, which will be v, and

the first value, so j would be Theta 1. We're just going to plot

the actual values and then mark those as the

true coefficient, we're then going

to plot the Theta path' we're going

to plot that path, again we're only using two

of the dimensions at a time. Let's say again, working

with b and Theta 1, then we'd plot the path of, if we set i equal to 0, then we want for the Theta

path only that first column, which will be the

different values for b, then for that second column the different values for Theta 1. If we set j here equal to 1. Then we can say what the initial value is

by calling Theta path 0, i and j and label that as the start,and also to note here, we are taking each

one of the steps that we take from the

start to the end, and using the dashed

lines as well as a marker for each step

of the triangle and then finally we're going to say negative one and i

and negative one and j for Theta paths in order to get the final value and

we'll label that the finish. Then that's just

going to be a subset, one we call plot all. What we're doing is we're

just taking each one of the different combinations

of axes that we can. We'll have v versus Theta 1, v versus Theta 2, and Theta 1 versus Theta 2. As we see here, we're calling plot i j 0,1, then plot i j 0,2, and then plot i j 1, 2 on each one of

our different axes. Then on top of that

we're also going to plot our loss factor to

see how we reduce the actual loss function as we iterate and move closer and

closer to the true values. With that, we get our

gradient descent to output both our Theta

path and our loss vector, we can then pass that

into our plot all function that we defined

with our learning rate, our number of iterations,

and our Theta. We run this. First, we have to obviously run what we

have here, we run that. This will plot out the

actual steps that we take. Here we see the Start and we see each one of these

triangles and are moving closer and closer. This axis here, if we look

at the top left plot, we see the x axis is Theta 0, the y-axis is Theta 1, and we want Theta 0 to be moving towards 1.5, that's our b. We want the Theta 1 to

be moving towards two. Then if we look at

the actual values, it looks like Theta 0

stopped that around 1.9 and Theta 1 also stopped

that around 1.9. Then if we look at

the top right graph, we see Theta 0 is

still the x-axis, so also still at 1.9, but now the y-axis is Theta 2, and that's actually pretty

close to five already. We can see each step it

takes along the way. Then here we can see

Theta 1 versus Theta 2, which started as we suppose all these

are starting at 3,3, then it should have

moved towards that 2 and 5 that we'd like. In the bottom right graph, we see the number of iterations and we see pretty quickly, once it gets to, it looks like maybe a hundred, two hundred iterations drops off to getting a very low error. That it slowly, very gradually continues to

minimize the error, but not quite as much, which is probably why

that learning rate, why we haven't gotten all the

way to our optimal values, and we have the gap between

some of our endpoints, our actual true values, and the endpoints that we

got using gradient descent. Now I quickly want to show

you what this looks like. If we were to decrease

the number of iterations, so if we decrease

this to say 100, and we run all this, we see that it stops a lot

earlier along the way. The gradient descent didn't

get to quite finish. Now if we keep this at 10,000, and we decrease

the learning rate, which allows for larger steps. We can see here, that it actually reached

the optimal solution. Decreasing the learning

rate allowed for those larger steps and

we were actually able to get to those

optimal solutions. Finally, I want to show

you what will happen if you set that learning

rate too large, now if we run this, maybe a little bit

difficult to see, if you see at the top left

corner of the top left plot, we see that we're talking

about massive numbers. It's 0.2 times 10 to the 305. We have missed our

value by a long shot. We totally over blew

the optimal value. If we look at the

actual error rate, we see as we increase the

number of iterations, the error shoots way up as we completely

miss the optimal value. That closes out our video here, working with vanilla

gradient descent. In the next video,

we will go through the same steps and briefly walk through how you can do it using stochastic

gradient descent.

Welcome back to our notebook

here on gradient descent. In this video, we're

going to close out by discussing stochastic

gradient descent. Rather than averaging

the gradients across the entire dataset

before taking any steps, we're now going to take a step for every single data point, as we discussed in lecture. Now, our exercise here

is going to be to run the stochastic

gradient descent that we have, this function, and then also modify the

code so that we can randomly reorder which one of

the different data points it picks at

each iteration, and I'll walk through

that in just a second. First for stochastic

gradient descent, much of the function

will look similar to our radiant descent function, we'll set Theta equal

to this initial Theta. We will have our zeros, which are going to

be the same size as not just here the

number of iterations, but also the number

of observations. Here, the idea is that the

number of iterations will be how many times are we going to go through the full dataset. But we're updating it

every single datapoint. By the time we run

through the full dataset, and our dataset has 100 values, we all have made 100 updates, so we have 100 iterations. If our number of

iterations equal to 100 and our number of

observations equal to 100, and then we'll be

making 10,000 updates, but only running through the

entire dataset, 100 times. This'll become more and more

familiar as when we work with our deep learning models

later on in this course. This will be something

called epochs, and the epochs will be the

number of times you run through the full

dataset, and then, you will also define the batch

size as we talked about, mini-batch gradient descent where we will actually find

that right balance of the batch size as well as how many times we want to run

through the entire dataset. So we're going to have

zeros for the Theta path, that's going to be the size

of number of iterations, times the number of

observations within our pass through matrix. We're then going to set Theta initial equal to the first value, and our loss vector again

is equal to the zeros for the number of iterations times the number of observations. Then, for the main stochastic

gradient descent loop, we're saying for i in range

number of iterations. But now we're also

saying for j in range, that the number of

our observations, so for i and then for j, and that will allow for every single value within

every single for loop. We'll do 100 iterations

and for each iteration, we're going to go

through each one of the different observations, and that's going to

be your value for j. Then, we're seeing to get

that gradient vector, so up until the gradient

vector, things are the same. For that gradient vector, we're going to say the value for j versus the value of j that was predicted

for that specific row, and then we're going

to take the x matrix, but only the jth row, in order to get

that.product to get our gradient vector just

for that single value. Then, with that new

gradient vector, we'll use that to update

our Theta values at each one of the different steps to get each one of our new Thetas. Then, we update the data path

according to the counts, and our count is being added by one each time we run through

either the a and the j loop. That's why we have this count

variable to keep adding on, rather than just if we were looking at the number

observations, 0-100, or just the number of iterations, which is also 0-100, the count will go from 0-10,000 as it will

have to go through both of these for loops. Then, we'll just return,

as we did before, the Theta path as well

as the loss factor. Here, we set the learning

rate to 1e\_negative 4, the number of iterations

here again is just 100, but that means we're

making 10,000 steps, but we're only running

through the full dataset 100 times and their Theta initial is going to be this 3, 3, 3 again. We can call stochastic

gradient descent, get our path as well

as our loss vector, and use that same plot all function that

we defined above, and plot out that datapath, the loss factor, and then the appropriate labels for

the learning rates, the number of iterations, and the initial Theta. We run this, and we can see

the path that was taken, and if we look at this

bottom left graph, that'll be your first

clear observation of the amount made that swerves back and forth

rather than creating a straight path towards

where we're trying to aim, as we did with the

normal gradient descent. So it's a bit more random. Now something to note is

if you are doing something like stochastic gradient descent or mini-batch gradient descent. Each one of our updates, as we go through each

one of the values, if we do that for loop

with that set ordering, the update for iteration 20 will be dependent

on the update from iteration 19 and the

update from iteration 18, so on and so forth. So it'll be biased according to the ordering of our

actual dataframe. Rather than do that, we're going to make this a bit more random, and at each time

set rad j equal to a random integer,

np.random.randint. Some value, the size of the

number of observations, so we can actually say zero

through num observations. We run this, and now, rather than that ordering, we had a bit more randomness, and we can see it's

a bit more squiggly, but that ensures that it doesn't have, like we saw before, clear pattern going

back and forth and it's not dependent on the

ordering of the dataframe. Now, we can play

around with this, as we see here, we can increase the

number of iterations. I'll run this because

this'll take a second. Now again, we're making 10,000 times 100 updates along the way. So using stochastic

gradient descent, hopefully we will be able

to get to that solution. But in general, using

stochastic gradient descent, as mentioned during the lecture, will allow you to

speed things up, but at the same time it may not exactly get to the right solution as it will be bouncing

around in order to get to that ultimate solution. That may have fake in

just a second to run. But as we see here, once we increase the

number of iterations, that may have been even

too many iterations. We didn't have to go

as long as we did. We see that we end up

at that final point, at the finish that

we'd hoped for with the Theta\_2 equal to 5, if we look at the bottom

left and Theta\_1 equal to 2. We see that we have this sharp decrease right at the beginning of the

number of iterations, and very slight amount of

improvement along the way, we do know where we

are aiming towards. Now something to note

here, as I said, we may not have needed to go as many iterations as we get towards the

bottom of that slope. If we imagine that concave curve that we discussed during lecture, the gradient is going

to get smaller and smaller as it approaches

that optimum value. So the updates will be

smaller and smaller as we get closer and closer

to the optimum value. That's going to ensure that if we have a small

enough learning rate, that we don't overshoot it, it just stays within that

right direction and keeps within that optimal minimal point within that concave curve. Now, that closes out

our section here on gradient descent,

and with that, we will get back to lecture

and discuss further working with our neural

networks. I'll see you there.

# Introduction to Neural Networks Demo (Activity)

Welcome to our lab here on

the introduction to neural networks. Here we're going to start off with an

exercise based on different logic gates, such as working with the AND

functionality or OR functionality. And then as we'll see later on, working with XOR in order to find if we

can actually use a single perceptron, as we discussed in the lecture,

in order to come up with these AND or OR functions, as well as these

more complex ones, such as XOR. So the first thing that we want to do is,

import our libraries, so we call import numpy as np,

matplotlib.pyplot as plt. And then we're going to

introduce the sigmoid function, which we discussed in lecture,

where we're just going to set sigmoid equal to 1 over 1 plus e to the -x,

where we can pass in values of x. And, if that value of x is very high, we'll end up with a value

very close to one. And if that value is very low,

we'll have value low, being in negative, and

we'll have a value close to zero. And as we saw with the graph,

it will range between the values of 0 1, but can't go any lower than 0 and

can't go any higher than 1. So we're just going to define

that sigmoid function, which is just going to be

1 over 1 + e to the -x. We're then going to plot it out,

and all we're going to do is say that we want 100 values equally

spaced between -10 and 10. And then we're going to take the sigmoid

of each one of those values, which we just defined above,

so that we get the activation. And then we're just going to plot

the different values on the x-axis versus the activation on the y-axis. And the rest is just drawing lines,

and creating a grid, and ensuring that we have the right y limits,

so that we don't go too far negative or

too far positive. So we run that. And we see that, as the value gets

close to -10, or essentially at 0, for our blue line, and

as our x-axis gets close to 10, then our y-axis gets

very close to positive 1. So with this in mind, and

how the sigmoid function works. With the idea that,

as we move higher, again, we get closer to 1, and

as we go negative we get closer to 0. Just to highlight as well at the point

of 0 itself, or at around 0.5. As you see, the grid line at 0.5 for

that y-axis, so that's going to be exactly 50/50 chance of either being 1 or 0,

if we were to create a threshold. Now, a logic gate is going to take in

two Booleans, so two different inputs. Either, usually true or false,

we're going to set to define true as 1 and false as 0. And then, with those two inputs

it'll return either a 0 or a 1, depending on the rule we defined for

that input. So we have here the true table for

a logic gate, that shows the output,

given that we're working with an OR gate. So if we think about an OR gate, if either

one of our two values are equal to 1, then our output should be true,

1 is equivalent to true. And the only time we should get false,

or 0, is if both of our inputs are equal to 0. So what we want to know is, can we come up with a neuron that

uses the sigmoid activation function, that we just defined that comes

up with values between 0 and 1, that will allow us to always output

the appropriate values of either 0 or 1? And the idea being that,

if the threshold is over 0.5, then the sigmoid would predict 1. If it's less than 0.5,

then it would predict 0. So we pass in x1 and x2 in each of those

will take on the value of either zero or one as well as the intercepts. And then we multiply each of

those by a certain weight. Now again, by limiting the inputs

of x1 and x2 to be either 0 or 1, we can simulate the effects

of this logic gate that we just saw in the table above

which which we saw over here. The goal is to find the weights

represented by the question marks we have here in this image, such that it

returns an output close to zero or one depending on what the inputs are. So the idea would be if we think

through the OR problem, if both x1 and x2 are equal to 0,

then we want to output a 0. Otherwise if either of them are equal

to 1, then we want to pass in a 1. So we have to think about

what those weight should be. And if we think about the plot that we

have above, if it's going to be very negative, again negative 10 or less, then

we would have a value very close to 0. And if it's very positive,

positive 10 then it's very close to 1. So that's our goal. So thinking this through,

you can see it the picture here below. We already have the weights, but

let's talk through how these weights will actually work in out putting the actual

value that you want for this OR gate. If x1 and x2 are both equal to 0,

the only value that's going to affect z, this equation that we have over here,

is going to be that intercept term of b. And because we want the results for

00 if both x1 and x2 are equal to 0 to be close to 0,

b should be negative. Has to be less than 0 to ensure that our sigmoid function outputs

a value less than or 0.5. Now if either x1 or x2 is 1 we

want the output to be close to 1. And that means that the weights

associated with x1 and x2 should be enough to offset

that -10 that we have for b. So if we give b that value of -10, W1 and W2 each have to be

at least greater than 10. So we set them each to 20,

so x1 is equal to 1. Then we have -0 + 20 positive 10. We pass that through

the sigmoid function and that would output a value very close to 1. Same would hold if we had x2

equal to 1 and x1 equal to 0. And then if both of them are equal to 1,

then we end up with positive 30. And again we get once we pass positive

30 through the sigmoid function, we value very close to 1. So as long as either x1 or

x2 are equal to 1, given the weights 2020 and

intercept -10, we have the value of 1. And if both of them are 0 then we

have the value of 0 pass through our sigmoid function. So here we see how we can come up with

the appropriate weights to ensure that we actually complete this OR functionality. So I run this. And the idea that we have here

is that we create a function for the logic gate that will take our

W1 an our W2 as well as our b. And then return the sigmoid of

w1 times x1 + w2 times x2 +b, which is what we hope to

ultimately output given that. We're passing in the z of w1x1

+ w2x2+ b into our sigmoid and then running the sigmoid and

then hoping for value of 1. Or 0, depending on what we want

outputs be if we're using an or gate or an AND gate, so on and so forth. And then we're going to

test it by saying for each one of these values 000110 and 1 1 we want to output for a and b. Given our test, what is going to

be the actual value of a and b. Again if we pass in that sigmoid,

then we will end up with a value that is 1,

once we pass 1 0 or 0 1. When we call mp. round, it'll start off, maybe with 0.9 and then we round it to ensure that we get 1. So we have our OR gate which is just

going to be equal to our logic gate which we defined as just the sigmoid

of that linear combination and we pass in our weights w1w2

ends are intercept of -0. We test that OR gate and

we get our different outputs. And we see that it matched up

according with what we saw our OR gate should actually be. Now let's quickly look at the end gate AND

how we can come up with the AND gate. So with the AND gate, when we look

at the table that we have here. If both of them are false, which is

our 00, then it should output false. If only one of them is true, then they're

not both truth that's the AND gate, right? With the and gate, you want both one input

and the second input to both be true. So we'd still have a 0 and it would stay

0 unless both the inputs are both true. So can we come up again with

the appropriate weights to ensure that if they're both true, then we end up with a truth value

otherwise we get a false value. So, as we see here,

we set the b equal to a negative value. That's negative enough that even once

we add on just one of these values, whether it's W1 or W2, that would be the equivalent of

just one of those being true. We still have a negative output. So, b plus W 2 \* 1 we'd still

have negative 20 + 10 and we'd end up with negative 10 as

long as it's negative our sigmoid function will output a value less than

0.5 and we round that down to zero. The only way that we end up with

a positive value is if both of these are true,

then we have negative 20 + 11 + 10. And then we'd end up with positive one. We pass positive one into

our sigmoid function and we have a value greater than 0.5. Now these W1s and

W2s can essentially be any number. Well first, let's show that this works. We see that if it outputs 0 for

every single value except for one one as it should with our and gate. We can see also if we wanted to,

we can make these values any value less than negative 20 or

less than the absolute value of 20. So that once it's added on,

it remains negative, but once both of those are added

on then it becomes positive. So both values have to add up to something

greater than 20 and be less than 20. So we can run this and see that

again we get all the correct values. Now we're going to do the same thing for

the M or gate and the M and gate and or an and just mean not or and not and so the

opposite of or and the opposite of and. And you'll see why this is important

once we get to the next exercise. So not or is just going to

be the opposite of the or. So if it's any of these three

values in this table which would have all been true for the or

then we set it to false. And we only keep it at true if

both values are equal to 0. And thinking through which weights

will work, we just need to ensure that we have a positive value if both are equal

to 0, otherwise we have a negative value. So we just have to ensure

that these are negative and their absolute value

are both greater than B. And then we have our an or gate. Will double check the Outputs and

we see that 00 is equal to 1, otherwise they're all 0. And then, finally, we're going to

closeout this video with an end. Where will see, again, this is just

the opposite of the actual end, so and would only be true if both

values were equal to true. Now that we do the opposite, it's true every other time except for

when both values are equal to true. So what we need to do is

we need to ensure that as long as we have

the Inputs both being 1, that will cancel out

the B that we had here. Otherwise we always have a positive value. So we do that by saying that

these two added together W1 and W2 will outweigh the B. Otherwise on their own,

they can never outweigh that B value, and that will ensure that we always have what

we have here in terms of the not end gate. And we can see that that holds as well. Now in the next video, we're going to pick

back up and discuss why there's a limit to only working with a single neuron and

how we can build off of a single neuron. Create another layer of neurons as we

do with our multilayer perceptron and come up with this X or functionality

which will discuss in the next video. Alright, I'll see you there.

Now, in this video, we're going to discuss the limits of

working with just a single neuron. So so far, in the past video, we saw all

the different ways that a single neuron would be able to handle coming up with

the AND gate, the OR gate, the NOT OR and the NOT AND gates. Now, we will see the limits when we work

with the XOR, or the Exclusive OR gate. And for those of you that have taken

computer science courses, perhaps, you're familiar with the XOR gate. For those of you that are not, the idea

of the XOR gate is to only pass through. If either one or

the other of our inputs are true, but not both of them make true, so

we see it both are false, we return false. But also, if both are true,

then we return false, only if exactly one of them are true,

do we return true. So can we create a set of weights, such as a single neuron can output

this property that we see here? And it turns out that we can't. And if we want to,

what we're going to need to do is actually create another layer, so

I'll pass in our input values of X1, X2, as well as our intercept. And then, we'll create another layer as we

do with our Feedforward neural networks, and see how using two layers we

can come up with this XOR gate. So the concept is,

if it's going to be XOR. We want one of the outputs

in the second layer, to actually be equivalent to the OR gate. And the other one to be equal

to the not AND, so the idea is, if we have, either one or

zero, so X1 equal to 1, or X2 equal to 1, or both equal to 1,

then the OR gate will return one. But it won't return one for,

the only one it won't return, one for us if they're both 0. And then, from the not ANDs,

it will return the true value or turn one for all of them, except for

both one and one being true. And then, we can take the outputs

of the OR gate, Andy, not AND gate. And in the second level,

add on another AND gate. And that will give us our XOR function. So if we think about that, if we start

with 00, then that will pass zero for the OR gate, so then we'd end up, when we

take the the AND gate at the second level, no matter what, if one of them is 0,

then we automatically end up with a 0, so that is correct,

in that both the inputs are zero, and with our XOR gate, as we see here

in the table, it should be 0. Now, if one of them is a one and the other

zero, the XOR gate will return a one, and then, not AND gate,

which will turn one for every single value, except for

ANDs, will both be 1. And in that second level,

if we take the end of both one and one, we will output one, so

will get the correct value. And then, finally, we want a 0. If both the X1 and

the X2 are equal to 1, are both true? And are not AND gate. If we have one in,

one will pass through a 0. So even though our OR gate would pass

through one, the AND gate at the second level of the one and 0, will end up

ensuring that it ends up passing is 0. So that will ensure equivalent to

the XOR final row that we pass out as 0. So again, the XOR gate will

just be a combination of an OR gate, and not AND gate. Which we learned just above, and then, we can pass that through in

the second layer with an AND gate. And that will output the correct

output for XOR gate, and we see that in practice where we define

our XOR gate as the combination of an AND gates of the output of both the OR

gate and the not AND gate, we pass in that cmd virtual pass

out the ones or zeros accordingly. Add on that AND gate,

and when we test that, we get the output that we would expect,

where we have the zero for the both of them being false,

and is 0 for both men being true. And then, true,

if just one of them is equal to true. So that closes out our discussion in

regards to working with that XOR gate, and adding on the extra layer,

and seeing how we can come up with more complex boundaries

once we move to multiple layers. Now, we discussed during lecture the

actual matrix weights taking our input, how that's transformed into

the first layer, and then, into the second hidden layer, and

then, eventually into our output. What we're going to do here is make

that more concrete by actually coming up with some random weights,

as well as some random inputs, and see how these matrix sizes transform, as we go

from our input through to our output. So here, we're going to start with

three weight matrices, W\_1, W\_2, and W\_3, representing

the weights in each layer. And the convention for these matrices is

that WIJ gives the weight from neuron I, from the prior layer,

two neuron J up until the next layer. So the weight of moving from I up until J. Now, a vector x\_in is going to

represent a single input as we discussed during lecture. We discussed just working

with a single input, as well as the full data set of inputs,

and x\_mat\_in is going to represent what a toy version of

a full data set with just 7 rose. And the goal for our exercise here, is

saying 4 input x\_in, or going to calculate the inputs and outputs to each layer,

as we move from our linear combination, which it outputs some Z value, and then,

taking the sigmoid of that value, and then, seeing how that's passed through

to each one of the different layers. We're going to write a function then, that does the entire neural network

calculation for a single input. Do that again for a matrix of inputs, and

then, test are functions that we just created, using our x\_in, and

our x\_mat, which is our toy data set. So let's look at this, W\_1, W\_2,

and W\_3, which will highlight for us how these actual weight matrices

should look in the back ends. Now, this isn't learning

the optimal parameters for us, but it's showing us just one step through the

feed forward from the input all the way through to the output, and

when we get back to lecture, we'll talk about how we can

actually optimize these weights. So here,

we start with a 3 by4 matrix for W\_1. We have three rows and

four different columns. W\_2 is going to be 4 by 4. And then,

column three is going to be 4 by 3, and while I say those numbers,

you should be thinking of how we're transforming our 3 dimensional

vector from X1, X2, X3. We take our 3 by 4 matrix to expand

that into a four vector, and then, keep that as a four vector by

multiplying that by a 4 by 4 matrix, and then, a 4 by 3 matrix to

ensure that we have three outputs. So that's the idea of W\_1, W\_2, and W\_3. So our inputs just going to be

these three values, .5, .8, and .2. And then, our toy data set,

which is going to have 7 Rose, is going to also have the three columns,

where we have as the first entry, the same entry that we have for x\_in, but

also seeing how this can expand to six more rows, or each row will have

the same amount of columns. We're defining here the softmax for

a vector, which is just going to allow

us to output probabilities, for a single vector, and then,

we do the same for a full matrix. So we run this, and we see our

output as mentioned that we have for our W\_1, 3 by 4 matrix. That's going to be multiplied

by a row vector, so 1 by 3 multiplied by a 3 by 4

should output a one by four matrix, so it should be our first hidden layer. And then, we see the matrix for

our toy data set, whereas, 7 rows. And if we imagine this is a 7 by 3,

and when we multiply this by W\_1. We end up in our first hidden

layer with Outputs for every single one of the different rows. Sole be a 7 by 4 matrix, and

we'll see this in just a second. So if we pass in,

let's first pass in here just the x\_in. And we take the dot product. Then, as mentioned,

that we will get the linear combination. Here, we're looking at Z2, which is going

to just be that linear combination, taking the broader product of x\_in and

our matrix W\_1, and we end up with this four

vector as mentioned. We're then just going to take

the sigmoid of that output, so we got the linear combination. Now, we take the sigmoid of that, and

we still have the same shape, but now, it's the sigmoid of each

one of those outputs, and now, that output will feed,

Into the next layer. And again that W\_2 was

a four by four matrix, and here we have a one by four vector,

so we'll end up with again, our z\_3 being a one by four vector. And once we have that linear combination,

we can again take the sigmoid and that will be the output

into the final layer. And z\_4 will be the dot product of A\_3 and

W\_3, where again, W\_3 is going to be that four by three

matrix to ensure that it matches up with the one by four matrix and

it outputs a one by three vector. And then we take that z\_4 and

call a soft\_max to see probabilities for each one of the different values. So we see that for the different classes that we predict

that the first class is the most likely. And that's the idea of feeding through

this neural network up until that soft\_max to come up with a complex

solution to our classification problem. Now quickly, I want to show you what this looks like

if we were to pass in the full matrix. So we run x\_mat\_in and

instead of it just being one row, now we have all seven rows

being passed through. And then we have that

seven by four matrix. We can take the dot product of that

output by the four by four and we still have that seven

by four matrix and then we take the sigmoid of the Z3. So still the same shape, but now taking

the sigmoid of each one of those values. We can then take the dot product so

that we can get the output of the linear combination of each of those, but

only outputting three different values. And we can take the soft\_max and

then we can see the probabilities for each one of the different values from the

output of that original matrix by taking the soft\_max. Now just to see how that

computes all the way through from beginning through to the end,

we create a function called the soft\_max\_vec which will

just be what we pass through the sigmoid of the dot product of x and

W\_1 and W\_2 and W\_3. And then we can do the same,

and just instead of passing in, it will all be the same

function as we just mentioned, the x will work just as well whether

it's a matrix or just an input. We do create two different functions and

we can pass that out, and we can see that we have

the solutions desired. All right, well that closes out

our video here working with, from beginning to end, a neural network. And once we get back into lecture, we'll

discuss how to actually optimize this model so that we're not just

looking at random weights, but eventually the optimal weights using

what we learned with gradient descent, and then something

called back Propagation. All right, I'll see you there.