In this section, we're going to go over

the basics of back-propagation, and that's going to be the magic

that ultimately makes training these complex neural

networks actually possible. So what's the process of

actually training a neural net? We saw before that we start

with some initial guess. We then saw as well how to get

a prediction by pushing that initial input through the feed forward network. The next step is to compare our

prediction to the actual value and calculate that loss function

J which measures our error. Once we see that error,

we can adjust our weights accordingly and repeat the process. So how do we know exactly how to

adjust each one of our weights? Back-propagation is ultimately going to be

that key framework that tells us how to make a single adjustment to our weights in

the right direction using the chain rule from calculus. So how do we train our model

before using gradient descent? We saw this a bit during

the gradient descent notebook. We made a prediction given

our initialized parameters. We then calculated the loss function for

that particular prediction, given the actual values we

were trying to predict. We then calculated the gradient of

that loss function with respect to our parameters. Recall that this will be the direction

of the steepest increase. By subtracting that gradient value

multiplied by some learning rate, we were able to move our parameters in

the direction that will minimize our loss function. And we iterate over and

over with the number of iterations and step ties pre-determined by our model. And ultimately, our goal is to reach the optimal values

that minimize our loss function. So let's start off our discussion of this

process for neural networks with these first two steps, making the prediction and

calculating the loss. So first we pass in our input values. We have initialized values for

each of the weights, and with that we calculate

the different values at each layer. Then we ultimately get our predicted

output values for that given input and the given weights. And with that we can

evaluate our loss function. For example, R-squared error or

are logistic loss. So how can we ultimately change our

weights to continue to lower that loss function at each iteration? Let's think of the neural

network as a single function F, that takes as input x and

output some value, y. The key to the complex computation that

makes up this function F, all comes down to the different weights, represented as

the different arrows in our picture below. And given the structure of our neural

networks where the inputs are defined by our datasets, and our activations will

be predetermined when we create our model. The parameters that we are trying to

learn are going to be those weights and ultimately, those weights

will define that function F. And ultimately, our loss function will

be a function of the true value y and this function F with the input of x. And if we focus in on just one layer, we can see how many different weights

actually need to be calculated. So to get the z value for

just a single node in that first layer, we'll need to learn the w's for each input

as well as value for b for each input. And then again another four parameters for

the second node of that layer, and through for

every node within that layer. And then we need to also do this for every node in all of the other

layers in our network as well. Now, our goal then,

when we find the gradients, is to find the partial derivative

of each weight in respective J. In other words, how much does a small change in

the parameter affect our loss function J? Now, given the way that

the gradient is calculated, this will tell us what direction to adjust

each weight Wk to lower our loss function. And once we're able to do this,

we can then just adjust and repeat. So that closeout this first video,

and then the next video, we'll officially introduce the concept

of back-propagation and how it ties with the idea of using the partial derivatives

to find these optimal weights.

Now getting into backpropagation, what we're ultimately

going to want, if you recall with the

gradient is going to be, that partial derivative in regards to each one

of our weights. Using that partial derivative, we'll be able to

update those weights in the correct direction

moving forward. Now this idea of being able

to use calculus to update our parameters is going to play an important role in how our neural network models

are actually constructed. Our functions used

to calculate our y, as well as our loss function, are going to be chosen so that they have nice derivatives, as we saw earlier when we touched on the derivative of

the sigmoid function. We'll ultimately need

to also be aware of some numerical issues

that will come into play when working

with these derivatives, such as exploding and

vanishing gradients, which we'll touch on later on. Now with that in mind, we're going to think of

the weights layer by layer and now going to dive a bit into the calculations used when actually

conducting backpropagation. So the values for the weights for that final layer in

our neural network will be updated using that partial derivative in

regard to the weights of that final layer and

that's going to be calculated by taking the.product of our error term y hat minus y and the output from the prior layer that fed

into our final layer. Then from there, in order to calculate the weights

for the second layer, the layer before the final layer, we take what we learned

from that final layer and take the.product of w

of that final layer, multiplied by the derivative

of the activation of z from that final layer. So this is again working with that final layer and with that the.product of the prior layer and that's our A1 over here. Finally, we add on the further steps needed and

take the.product with x, our initial input in order to get the derivative in respects

to our initial layer. Notice how these will be affected by our actual error term. So the larger or

smaller or errors will affect the size of

each one of our gradients. Also note that if we use the

sigmoid activation function, that the derivative is

the simple sigmoid of z times 1 minus the sigmoid of z. We're gonna touch on

this later on when we talk about the

vanishing gradients. We will be using this in our notebook and

although it looks a bit complex and may have sounded a bit complex as

we talked through it. We'll see you in the

notebook that they're actually quite easy to compute. So the idea of

backpropagation is that we'll first run our neural network with our initialized weights. Then moving back

through our layers, we're going to take the

derivative of each of our weights in our final layer with respect to

our loss function. Then use that to again get our partial derivative

in respect to our layer 2 of our weights and then our

layer one weights finally. We'll use these to update our initialized

values and then again feed these updated

weights through our neural net and

repeat the process. Now I want to quickly

touch on this concept of vanishing gradients

that I discussed earlier. Recall that this was the

derivative, what we see here, that we get for updating our three-layer feed-forward

neural network. What I want to highlight here is the fact that

we are multiplying the derivative of the

activation function of our z. So the derivative of that

activation function of our z from all the other

layers later in the network. Now with that, we want to re-emphasize that our

sigmoid function. For our sigmoid function, the maximum value

the derivative can take is going to be 0.25, and you can run the math, but this is due to the fact that the sigmoid can only take

on values between zero and one and thus the max

value it can take on would be 0.5 times 1 minus 0.5 when we talk about

the derivative here, and that would equal our 0.25, which we're stating

is the maximum. Now if we think about this, the fact that 0.25 is

the absolute maximum. If we continue to make our

network deeper and deeper, and we continue to multiply

by these small values, the gradient at these

earlier layers, such as the W1 that we see here, will eventually get

very, very small. This problem of the gradient eventually getting

incredibly small as we create these deeper

neural networks is what we call the

vanishing gradient problem. For this reason, other activation such as ReLU and others, which we'll touch on right

right our next notebook in the next video, have become more and more common.

Welcome to our demo here

on backpropagation. In this exercise, we're going to use backpropagation to train a multilayer-perceptron or our

feedforward neural network that we've gone over in lecture, using just a single hidden layer. You'll have an

opportunity on your own to play around with

different patterns. We're just going to

focus on one pattern here doing a

classification problem, and you'll see that

in just a bit, and see how quickly or slowly the different

weights converge. With that, we'll

also see the impact and interplay of

different parameters, such as the learning rate, the number of iterations, and the number of data points

that were working with. First thing, we import our

library and then just to give you a overview of what we're going to go over in this lesson. In this exercise, we're going

to prepare a code to create a multilayer perceptron with

just a single hidden layer, and in that hidden layer

they'll be four notes, and train it using

that backpropagation. If you notice in the libraries

that we just imported, we're not going to be using the traditional deep

learning libraries just yet. We're going to show you

just using NumPy how to go through these steps

of the feedforward, and then the backpropagation in order to learn the

optimal weights. In order to do so, we're going

to initialize the weights to some random values between

negative one and one, as we discussed again in lecture. We're then going to perform that feedforward computation

through our neural net. We'll compute that loss

function at the end. Using that loss function, we'll calculate the

gradients for all of our weights using

backpropagation, telling us in which

direction to update the different weight

matrices using also our learning rate

parameter which we'll define. Then we'll execute

those steps 2-5 for a fixed number of iterations that we'll define as well. With that, we'll

plot the different accuracies and the log loss, and observe how they

change over time. We'll be using the log

loss here since we'll be doing a

classification problem. Then once the code is running, you're going to address

the following questions: Which patterns was the neural network able to learn quickly and which

ones took longer? Again, you can go through

this on your own. What learning rates and number

of iterations worked well? Then if you have time, you can try varying the size of the hidden layer and experiment with different

activation functions, including ReLu that we have here, which we'll discuss when we

get out of this notebook. That might take a bit

more to play around, but that's definitely an

option that should be available as you get more and more familiar

with the code. With that, I'd like to walk through the code

that we have here. We're going to start off with a certain number

of observations. We're going to create our

initial pattern here. We're saying there's going to be 500 different observations

in our dataset, so equivalent of 500 rows. Then our x matrix is just going to be random values between

negative one and one, and we're going to have

two different variables for our x matrix. These are the two variables

for our input values. With that, we're also going

to predefine that bias term, which is just going to

be a bunch of ones. We concatenate

those two together, and that's all the values that we need to learn weights for. Those are the actual values. For each one of

those actual values, we will learn the

optimal weights given the two x values as

well as that bias term. If you'd like to do

a different pattern, you can just uncomment any one

of the different patterns. We've done the diamond here. Each one should work fine and you can look through

each one of them. We're going to print

out the actual shape to see what this looks like. But if you think about

the diamond pattern that we are outputting, we have values between

negative one and one, and in our x\_mat\_full. We're going to say we want the absolute value of

each one of these, and if it's less than one, then we will mark

it a certain value. Here, where you would

mark that as true. Then anything greater

than than one would be false or zero and one once we make

that astype integer. That will allow us to classify whether or not it's in

the diamond or not. If you think about the

absolute value being greater than one when

they're added together, for random values between

negative one and one, that means that both of our first two x values

have to add up to one; the absolute values of them. If it's greater than that, so foreign value's 0.75 and

the other one's negative 0.5, then that would be outside of the range because it's

greater than one; the absolute value of that sum, and we would be

outside the circle. We'll see this in the

graph in just a second. We're saying what the actual

shape is of our x matrix, as well as our output y, which should just

be one-dimension. X\_mat\_ full should be 500 by three since we're

including the bias term. Taking that x\_mat\_full, we're

going to plot these values, where our y-value that we

defined up here is equal to 1 and we will mark

those with Os in red. Then for those that

the y is equal to 0, so we're taking a

different subset, we'll mark those with x. So that we can see that diamond

that we're talking about. You run this and you can

see our diamond here. You see as those values

stay less than one, so this is essentially, if you look at the bottom

or the top corner, you see that or any

one of the corners, those are going to be the

values of one and zero. Then there should

be approximately a straight line dividing between those that are less

than one and those that are greater one when you take the absolute value of each one of the x and y components here or the x1 and x2 components here, and sum them together. We want to come up with

this classification method using what we learned in

regards to our neural networks. Clearly, it's not going

to be a linear divider, it's going to have to be a

bit more complex than that. In the next video,

we'll walk through how we initiate each one

of the functions that we need as well as that feedforward and

backpropagation structure, and start to learn the actual

weights that we need in order to classify this object. All right, I'll see you there.

Welcome back. In this

cell that we have here, we're going to walk through a bunch of the

different functions that we're going to use throughout in order to make

that feed forward neural net, as well as take those

back propagation steps. To start off we define

our sigmoid function, which you should recall from

lecture is just going to be 1 over 1 plus e to

the negative x, x just being our input here. Generally speaking,

we'll be passing in that z value in

order to get our a, if you recall the symbols that we use throughout

the lecture. Then we're going to

define our loss function, passing in y\_true and y\_pred. We also have this epsilon, which is going to be

this very small value, and the reason for that is just that we will tend to error out if we get exactly 1 or

exactly 0 for our prediction. As we see here, if our prediction

y\_pred is equal to 0, we'll take the maximum of

that value and our epsilon, which is this incredibly

small number, and we'll end up with that

incredibly small number rather than ending up a 0. On the other ends, when

we take the minimum between y\_pred and 1

minus the epsilon, if y\_pred was exactly 1, then 1 minus the epsilon

will be 0.99, etc. something very close to

1 but not exactly 1. Then once we have that we're going to compute the

log loss function, which is just going to be the negative log of our

prediction times y\_true, and if you think about y\_true, it's either going to be 0 or 1. If this y\_true is equal to 0, and you end up predicting 0, then np dot log of 0 would

be 1, but this cancels out. It'll be very low,

that arrow will automatically cancel out and

it won't count for that. But if this value

is 1, y\_true is 1, and you predict something

very close to 0, then you'll end up

with a higher value. The same thing goes

on the other end. If y\_true is equal to 1, then this will cancel out. But if it's equal

to 0, then it will maximize this error

because the log of 1 minus 0 will be the log of 1 and have

that maximum value. That's our loss function, and that's going to be the

log loss again because we're doing classification

between 0 and 1. We then want to define

here our forward pass, and we're going to pass in

our initialized weights, W1 and W2, as well as our

updated weights throughout, and we'll see how we do

this in the next cell. With that we're

actually also going to be doing our

back-propagation steps here. We're also going to be

computing the gradients. We're going to use the output of this to do our

back-propagation steps. We're pulling in these global

values that we've defined, x\_mat, y, and num. We're then going to compute the new predictions

step-by-step here. If we think about our

feed-forward neural net, in order to get to z\_2, that next step in our process, we just take the dot product of our matrix of our

original inputs and W1. That'll give us z\_2. In order to get a\_2, we have to do our

non-linear transformation, which is going to be our

sigmoid function on that z\_2. First we did the linear step, then that non-linear

step taking the sigmoid, and then z\_3 is just going

to be taking that output, and z\_3 will ultimately

be the last step. We start to take the sigmoid

before our prediction, because we're only

doing one hidden layer, and we take the output

from the prior layer, which was a\_2, and we take that with a

dot-product of W\_2. Then our prediction

is just going to be the sigmoid of that z\_3. When we reshape it here

we're just making sure that it's only one dimension. Now to compute the gradient, given the loss function

that we have defined, our loss function in respect to z is just going to be

negative y plus y\_pred. Now this is in respect to z. Before when we were looking

at loss function in the lecture we were saying in respect to that final output, which would be a\_3, but here we're starting with z\_3. We take that output of Jth of the gradient of

J in respect to z, and we use that in

order to calculate the gradient of J in respect to W. That's just going

to be this J\_z\_3, that prior gradients

that we just calculated, that and the dot-products of a\_2. Here we're not calculating

it in respect to J, or calculating the gradient

of a in respect to z, and this just is going

to be, if you recall, what we do with the z

value in order to get a, we're just going to see, is going to be the sigmoid of z\_2 times 1 minus

the sigmoid of z\_2, and that's just going to be again that derivative

of the sigmoid. Then we're going to use

everything that we've calculated so far in order to get the

gradient in respect to W\_1, which is just going to be the

dot product of that J\_z\_3, which is what we defined up here. We reshape it just

to insure it's in the right shape and that it's not a one-dimensional array. Take the dot product of that with W\_2, transposed

specifically. Then we multiply it by this value that we

have output over here, and then we take the dot

product of the transpose of that with our original input, and the transpose of

that original input. We just go back through

each one of the steps, taking each one of the different gradients

that we learned before, using back-propagation

to ultimately get the gradients in respect to each one of our

weights: W1 and W2, which is going to be what

we need in order to do our back-propagation steps and continuously update each

one of our parameters, which are just going

to be our W1 and W2. This function will return

both our prediction, which we have defined up here, as well as this gradient, which is going to be this

tuple that we have here. That gradient's going

to have the gradient in respect to each one of

our different weights. Then finally we define this plot loss accuracy which will just

show us each one of the different loss values

and the accuracies, and we create our initial figure, we say what the title

is going to be, and we're going to

create a sub-plot. We're going to create

two different plots, one for the loss, one for the accuracies. Ideally the losses

should be going down and the accuracy

should be going up, and we're just going

to call ax dot plot, so it's a very simple plot. Then we're going to do that

again for our accuracies. We have our loss values

and our accuracies, and those are passed

into the function. Now we see all of the different functions

that we have available to us, and we're going to use those in the next video in order to do both our feed-forward steps as well as our back-propagation, and then ultimately once we do that and go through a

number of iterations, see the output of the

different accuracies and the different losses.

I'll see you there.

Hopefully, given

the functions that we worked through

in that last cell, you've been able to

think through how we would move forward

through our neural net, as well as how we'd use

that back propagation and the output from our function specifically from

that forward pass, given our gradient

and our prediction. How we would actually go

about using those outputs, iterating over a

number of iterations, and updating our model. With that in mind, hopefully you were

thinking through it, we're going to walk through

that code over here. The first thing

that we want to do is create our W\_1 and W\_2. Again, those are just going to be random values here to start. They're going to be random values between negative one and one, and if we think about

the weights for W\_1, it should be a three

by four matrix, since our input should

have three-dimensions, our two dimensions, X\_1 and X\_2, as well as our bias term. We're including that in here. Then it's going to be by four since we'll

have four nodes. Then that means that our W\_2 will have to just

be of size four. We are just going to

have four because that's going to be our

output after that. When we take the dot product

of our output with W\_2, then we'll be able to output just the actual values of a classification value

of either one or zero. We're going to set the number

of iterations here equal to 5,000 and our learning

rate equal to 0.001. Then we're just calling X\_mat, the X\_mat full so that

we can pass it through. Our loss vals and our accuracies are

going to start off as empty lists as we have here and then we're going

to do our iterations. For i in range number

of iterations, 5,000 different iterations

we're going to take our Y\_pred and our output gradients

from our forward pass. I talked about this just

in the top of this video, that this forward

pass will output both a prediction as well

as our gradient values. That gradient value output, if you recall from last video, it's going to be a tuple

of two different values. We're going to pull

them out separately as J\_W\_1 grad and J\_ W\_2 grad. Then to update the weights, all we have to do is take that initialized value

after the forward pass, which we've done with

our forward pass and gotten our gradients, we can subtract out the learning rate multiplied

by that gradient, to take that small step in the right direction in

updating both our W\_1 and W\_2. We can then get what

our current losses by calling loss-function

on Y and Y\_pred. We then append that value to the loss\_vals and then

our accuracy is just going to be how many we predicted correct over the total

number of observations. Just because we're using

the log loss again, our prediction will be some

value between zero and one. It won't be exactly zero or one, so we just are going to make them discreet values

of zero or one by setting Y\_pred greater than

or equal to 0.5 equal to one, and all the values

lower or equal to zero. Then we can append to our list of accuracies that value back. That's just for our

first iteration. Then what we have here is that we're just going to print out at every 200th iteration on what the log loss was and

what the accuracy is. Then at end, we can plot that loss accuracy

once we've gone through every single iteration

of those 5,000 iterations. Now coming back to

this W\_1 and W\_2, we think about the

fact that we have updated this after

the first iteration. Once we update it, we can then pass it back into this W\_1 and W\_2 for

the forward pass, since we updated it as itself, and we continue to update

that and get closer and closer to that optimal value. We're going to run this and

we can see the outputs as we do different sets of 200 values, 200, 400, 600, 800 and so on. We can see that that log loss goes down further and further, and our accuracy goes

up proportionally. As we go through each one of

the different iterations, and we see here at the end, we end up with 4,800 iterations

and an accuracy of 94.4. You can play around with

changing the learning rate. If you imagine I made that

learning rate real small, I want you to think

about what would happen. I'm going to run this and we see that it is updating a bit too slowly as we set that learning

rates incredibly small. We don't want to set it

too small or too large. We'll set this back

to what it was before so that we can

do the next step, which was 001. Then we can actually plot out where we got it

correct and where we got it incorrect to see where on our diamonds we were more

likely to have errors. We run this and these

plots should be simple enough given everything

that we've learned. We see the false positives and false negatives all tend to

be right around these edges. We did a pretty good

job of actually finding net

classification boundary. It's just right along the edges, perhaps some are

correct or incorrect. Now feel free to play around with different shapes to look at

what type of errors come out. You see everything that we can do here within this notebook, playing with the different

number of iterations, playing with the learning rate. Then we discussed also

that you can play around with a different

activation functions. Here we have it

defined as Sigmoid. You'd have to pass in

something else besides sigmoid, and with that in mind, that's a smooth

transition into what our lecture is going to

be in our next lecture, which is going to be discussing different

activation functions. All right, I'll see you there.

Now let's introduce some

other activation functions besides just working with the sigmoid function that

we discussed earlier. Let's go over the learning

goals for this section. In this section,

we're going to cover the different

activation functions for our nodes within

our neural network. Then at the end, we'll

summarize each one of the commonly used activation

functions that we discuss. Recall that this basic

perceptron that we have here is the simplest

building block for our neural network. The classic perceptron

model typically used a step activation function, so zero for values less than zero and one for

values of above zero. As we get that input, that input z into our

activation function. Now, the activation function

allows more flexibility than just this zero-one step

function, as it's called. Without this activation

function in general, the model would be

a linear model. The activation

improves our ability to determine non-linear outcomes. In the next slides, we're going to discuss

various activation functions. A great analogy, building off of our basic statistical knowledge and what we have

discussed thus far, is that of thinking of

logistic regression as linear regression with a

sigmoid activation function. Recall that we had that

linear regression and that linear combination

of those paths weights, and then we pass that through a sigmoid activation function. That's the same as working

with our logistic regression. In this analogy, we use the sigmoid because we want

outputs between zero and one, and we want a non-linear model. With deep learning models, we sometimes want

more flexibility over which types of

outputs we can consider, and we may not only

want to emphasize positive values between

zero and one, for example. We're going to start

our discussion of activation functions with

that sigmoid function, and this is the only

activation function that we've discussed thus far. Some advantages of this

activation function are that it produces that simple derivative that we saw earlier, and that it keeps the values

between zero and one, and technically, never gets

to exactly zero or one. As it goes to infinity

or negative infinity, it gets closer and closer, but not exactly to zero and one. The disadvantage and

the disadvantage that we discussed earlier, and something that we can

even see graphically here, is that the derivative can

tend to be a very low value. Now what do I mean by the fact that we can

see this graphically? The derivative is

meant to show us how much y changes with

a tiny change in x. If we look to very small

or very large numbers, we can see that the y value

barely changes at all. For x values between 5-10 or between negative

five to negative 10, we can see that the y values

are essentially flat. This will get even worse for x values beyond 10 or

beyond negative 10. The derivatives are going

to be very, very small. Although the sigmoid

function is easily interpretable and keeps

values between zero and one, it's very prone to this

vanishing gradient problem, and thus will often lead to

difficulty when trying to optimize using gradient

descent and backpropagation. Next we have the hyperbolic

tangent function. Again, we're going through our different

activation functions, which is the equation

that we have here, which is tanh of z, which is equal to sinh

of z over cosh of z, which is just equal

to e to the 2x minus 1 over e to the 2x plus 1. Now, what's important here

is not the equation itself, but rather some of the

properties of this function. In a lot of ways, it's

going to be very similar to the sigmoid function

that we just discussed, just a bit stretched out. Rather than with the sigmoid, where the sigmoid of

zero is equal to 0.5, the tanh of zero is going

to be equal to zero. As we approach infinity

and negative infinity, we approach one and negative one rather

than one and zero, as we did with a

sigmoid function. What does this look like? When we look at the

graph of this function, and we think about it in relation to the sigmoid function, we see that for values

between negative two and two, we have a sharper slope. Thus the derivative is

going to be a bit larger, i.e., a small change in x

equals a larger change in y. In gradient descent, may

be optimized or work a bit faster when working with

this activation function. It's powerful if, for any reason, you want your values to

be between negative one and one rather than

between zero and one. But as we discussed with

the sigmoid function, and as we can see

here on the graph, we still have that

same problem of very small derivatives for

a higher absolute values. Thus again, face that

vanishing gradient problem. So in order to answer this

vanishing gradient problem, we have introduced the rectified linear

unit function or ReLU. This activation function

is actually quite simple. For any z value less than zero, we just return the value zero, and for any z value

greater than zero, we just return that

actual value z. So we're essentially taking the maximum value between

that output z and zero. So really, if zero is

again going to be zero, ReLU for any value of z greater

than zero is going to be equal to z and the ReLU of any negative value

will just be zero. So what does this look

like graphically? We can see here here and

thinking through this graph, we can see that it is

still non-linear as we see this transition between

less than zero and greater than zero introducing

that non-linearity. As we can see on the

right side of zero, we no longer have that tiny derivative causing us problems. Then on the left side, rather than those tiny

changes, there's zero change. So these values will actually

zero out particular nodes. Now, this zeroing out will

allow for us to ignore nodes that may not be providing

much extra information. Thus it may be more

efficient than the sigmoid or hyperbolic

functions that always maintain at least some

information at each node. Now, on the other hand, there will be no learning happening at each of those

nodes that are being zeroed out and

perhaps you want to ensure some type of

learning at all nodes. With that in mind, we have the leaky rectified

linear unit or LReLU. The way that this works

is for positive values, the function remains the same. So it's just going to

be z as your output. But for negative values, rather than simply zeroing out z for z values less than zero, we multiply that negative value, that value z, by

some small number. That is the new output rather

than just having zero. So now our function

is going to be the maximum value between z, as we had before, and then rather than zero, sum Alpha less than

one multiplied by z. Recall that this is

going to be smaller than z itself if we're working

with negative values. So any negative number

multiplied by a value less than one will be larger than that

original negative number. So our outputs for

LReLU are going to be zero for z equals zero, z for any value z

greater than zero. Then for values less than zero, for z values less than zero, it will be Alpha

multiplied by that z. So what does this look

like graphically? Now we can see with the

graph for this function, we no longer have that zero

value at any of our nodes. This will solve that problem of nodes zeroing out

throughout our network while keeping the advantage of a steady learning rate without that vanishing gradient problem. Now, I would like to note here, just because it solves a potential problem that

regular ReLU may have, this doesn't mean

that leaky ReLU's are necessarily better

every single time. They are better a

lot of the time, but they aren't necessarily

better all the time. Sometimes you may want a more efficient

network that allows for zeroing out of some nodes. The best practice

would be to try both, starting with ReLU or

leaky ReLU and then trying the other as well and saying which one

performs better. So to summarize, the sigmoid function is

going to be powerful when we want outputs between

zero and one but will again suffer from that

vanishing gradient problem. The hyperbolic tangent is useful if you want

outputs between negative one and one and perhaps a bit

of a steeper slope, but also suffers from that

vanishing gradient problem. ReLU will solve the

vanishing gradient problem, but potentially suffers from

that dying ReLU problem. Dying ReLU problem is what it's called and that's just that

zeroing out of certain nodes. Finally, the leaky

ReLU will also solve that vanishing gradient

problem that was introduced with sigmoid

and hyperbolic tangent. But also solves

for that potential of the dying ReLU

problem as well. So to recap, in this section, we discussed the different

activation functions that we just went

over for nodes in our neural network and summarized each one of those commonly

used activation functions. Now in the next video, we'll begin to introduce a topic that will be important for any machine learning problem but especially so for

deeper neural networks. That's going to be

fighting over fitting by introducing different

regularization techniques. All right. I'll see you there.

Now we've discussed the importance of

regularization and the trade off between bias and variance when working with

different algorithms in prior courses. In this current set of videos, we'll discuss different regularization

techniques for deep learning specifically. And why it's of utmost importance when

creating these more complex models. So in this section, we're going to

cover regularization techniques for deep learning and why it's so

important for deep learning specifically. Will touch on dropout as

well as early stopping, which are different regularization

techniques specific to neural nets. And what that will also discuss different

approaches to optimization when working through our neural nets and

why those are important. So to start off, technically a deep neural

net is a neural net that has two or more hidden layers and often many more. So a neural network becomes a deep neural

network once we move beyond two layers. The importance of regularization when

it comes to deep neural nets is that with more and more layers we can

learn more and more complex models. And those complex models can often nearly

perfectly fit to our training data. But due to its complexity will often

overfit to that training data and not generalize well to that new data. Now with that in mind, let's quickly

remind ourselves what regularization is. Regularization is any modification we make

to a learning algorithm that's intended to reduce its generalization error, but

not necessarily its training error. So again, we're not optimizing

to training error, but rather tweaking our algorithm to

optimize her generalization, and usually at the price of

additional training error. So for neural nets, there are several

means of regularization available to us. Adding some regularization penalty in our

actual cost function is an option and this would be similar to what we saw

with Lasso or Ridge for example. Where more and higher weights will be

penalized within our actual cost function. We have something called dropout where

we'll randomly lose certain neurons in our network to ensure no model is over

reliant on any particular neuron or any particular path. Talk about early stopping,

will just be the idea of stopping gradient descent short, so that's not

perfectly fit to the training set. And then with that, to some degree,

those ideas of stochastic and mini-batch gradient descent that we

discuss in prior videos may ensure that we don't perfectly fit to our training set. And therefore, may generalize a bit

better than full-batch gradient descent. Now starting with that first option

that we had listed there, we can choose a loss function that will penalize

our model for having higher weights. And again, this will be similar to Ridge

regression if we're trying to predict a numerical value in that we can take

something like R mean squared error and add on a regularization term that

penalizes the weight squared. And if we are trying to predict

a categorical variable rather than a numerical variable, this will be very similar in that we

will be simply adjusting our categorical loss function that we have to

penalize again those larger weights. Now let's move to method that will be more

specific to working with Neural Networks, namely dropout. Now with dropout we'll be randomly

removing a subset of the neurons for each batch. So a common problem that neural nets

face in the earlier implementations was that although there

are all these different pathways available when optimizing

these complex neural nets. Often they would become over

reliant on particular pathways. Well, not much learning was done on

other pathways through other nodes. So creating a deeper or denser network,

adding on more layers or adding on more nodes, would tend to not

add that much value to our neural net. By adding in dropout, our neural networks will not be able

to over rely on any individual pathway. And thus it will become more robust to

over fitting to that particular pathway. And finally, since were randomly dropping

out neurons and learning on just a subset, we then will have to re scale those

weights of the neurons at the end to reflect the percentage of the time

that that neuron was active during training versus not active. So we can think of this image that we

have here as our standard fully connected feed forward neural network that we

are familiar with at this point. Would dropout, at each iteration we end up with

something like we see here to the right. Or we drop a predetermined percentage of

the nodes at each one of our layers so that our model is forced

to learn iterations with different possible pathways

through to our solution. So what exactly do we mean or so when we say that we are re scaling

our neurons after training our model? If you think about it, when we are training our model our node

is going to have a probability p as we see here to left of not being

present at any particular iteration. And thus the remaining weights are going

to be scaled up to make up for that. Thus, when we actually get to testing

time in all the weights are present, we need to ensure that we appropriately

re scale our weights, and that's why we have p times w at the end

when all these weights are always present. Now another heuristic that was available

to us that we discussed earlier this idea of early stopping. And this just refers to choosing some

rules at which to stop training our model to ensure that we don't over fit. Now an example of this would be to start

by checking the loss on a validation set. So not the training set, but rather some holdout set that

validation set at every 10 epochs. And if the validation loss at that

next step 10 epochs later is higher than it was at the previous step,

then we would stop training. So that's the idea of early stopping. Now that closes out this section on

common regularization techniques that are available to us. In the next, videos, we'll discuss another

important part of tuning your neural net, namely what type of optimizer

you're going to use. All right, I'll see you there.

# Summary/Review

## **Introduction to Neural Networks**

Neural Networks and Deep Learning are behind most of the AI that shapes our everyday life. Think of how you interact everyday with these technologies just by using the greatest features in our phones (face-recognition, autocorrect, text-autocomplete, voicemail-to-text previews), finding what we need on the internet (predictive internet searches, content or product recommendations), or using self-driving cars. Also, some of the classification and regression problems you need to solve, are good candidates for Neural Networks and Deep Learning as well.

Some basic facts of Neural Networks:

* Use biology as inspiration for mathematical models
* Get signals from previous neurons
* Generate signals according to inputs
* Pass signals on to next neurons
* You can create a complex model by layering many neurons

The basic syntax of Multi-Layer Perceptrons in scikit learn is:

**# Import Scikit-Learn model**

from sklearn.neural\_network import MLPClassifier

**# Specify an activation function**

mlp = MLPClassifier(hidden\_layer\_sizes=(5,2), activation= 'logistic')

**# Fit and predict data (similar to approach for other sklearn models)**

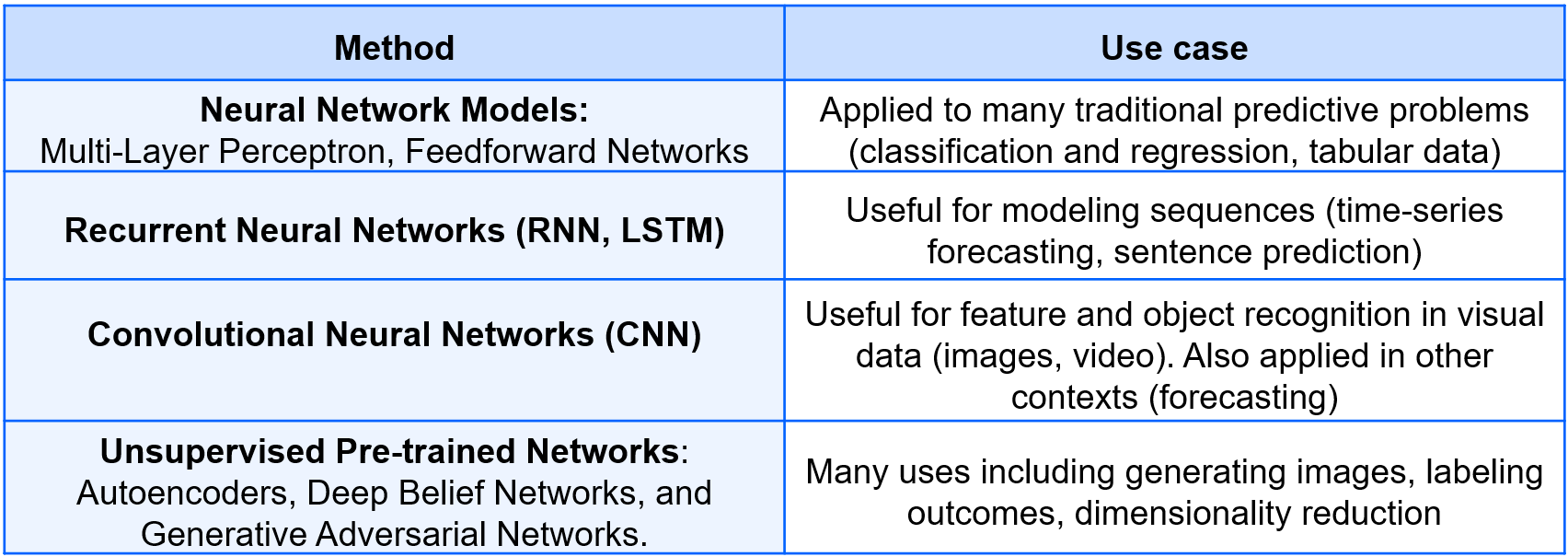
mlp.fit(X\_train, y\_train)

mlp.predict(X\_test)

These are the main parts of MLP:

* Weights
* Input layer
* Hidden Layer
* Weights
* Net Input
* Activation

## **Deep Learning Use Cases Summary**



## **Training a Neural Network**

In a nutshell this is the process to train a neural network:

* Put in Training inputs, get the output.
* Compare output to correct answers: Look at loss function J.
* Adjust and repeat.
* Backpropagation tells us how to make a single adjustment using calculus.

The vanishing gradient problem is caused due to the fact that as you have more layers, the gradient gets very small at the early layers. For this reason, other activations (such as ReLU) have become more common

The right activation function depends on the application, and there are no hard and fast rules. These are the some of the most used activation functions and their most common use cases:

