Now, let's talk about optimizers. So in this video, we're going to discuss

different optimizers available to us when learning the appropriate weights for our

given dataset and our neural net model. So far we've discussed

different approaches to gradient descent

that vary the number of actual data points involved in each one of

our steps in our gradient descent steps. Such as a single data point in

stochastic gradient descent, a subset of data points with

mini batch gradient descent, an entire set with full

batch gradient descent. Now, no matter what we use, they all had that same update

formula to find the optimal weights. Our weight at the next iteration

is going to be equal to the prior weight minus the gradient

time some learning rate alpha. But there are actually several variants

to the step of updating the weights that will give us better performance. And these tweaks to the updating step will

all be built around improving further and further from this original

formulation that we see here. And these different methods

of updating the weights or optimizing these weights

are going to be called optimizers. So let's start with

the concept of momentum. With the regular gradient descent,

you'll generally move slowly towards your optimum, and you can be changing

direction fairly frequently. Now, with momentum,

you're going to smooth out this process. And you do this by taking somewhat of

a running average of each of the steps and the smoothing out that variation of

each of the individual steps for regular gradient descent. So if we look at our formula, we see that

rather than just updating our weights with that gradient, we also look back to

prior values to smooth out these steps. So this is the at step t that we have, will incorporate some amount

of V at time or at step t-1. As well as the current gradient

at the step that we are at. With this in mind, our n value here is going to denote

our momentum hyperparameter. And the larger the value is for

that momentum hyperparameter, the more we are going to be

smoothing out our values. In other words, the more we are incorporating past

values into our running average. And will be giving values

less than 1 in general and a common value chosen

here is going to be 0.9. But again, if you want smoother steps,

use a higher value, otherwise use a lower value. Also worth noting, if you want to look

at perhaps further reading on your own in regards to momentum, often that term

n is going to be replaced by beta. So beta is going to be

the common nomenclature for that value and

the alpha is replaced by 1 minus beta. So n is going to be replaced by beta and

the alpha that we see here that were used to using as that learning rate

is going to be 1 minus beta. And when we choose our n and

our alpha in practice, we may want to keep in mind

using this relationship. So if we choose and an equal to 0.9, you'll probably want to

use an alpha around 0.1. So just to show this

in terms of a picture. For gradient descent, we can see that we take small steps

that can fluctuate quite often. Now, with momentum,

we tend to smooth out those steps. The fluctuations aren't

going to be as dramatic and the steps in get much larger

as momentum is gained. Also worth noting, is that momentum can cause you to

actually overshoot your optimum value. But the momentum will shrink at this

point, and you should be able to come back to that optimal value as

we see here in the picture. So the idea with Nesterov Amentum,

which will build off of the momentum we just learned, is going to be that

it will look and control for this problem of overshooting, and

they'll do so by looking one step ahead. So now,

rather than just taking the momentum and taking into account the gradient at

the current step, we take the momentum and the gradient at the step with

that momentum accounted for. So you see, rather than just

taking the gradient of the cost function as we did before,

we take the gradient of the cost function with n times Vt- 1

timestep before accounted for. And this will work because

generally speaking, the momentum vector will be be

pointing in the right direction. So be a bit more accurate to use

the gradient with a momentum accounted for than the gradient at

that original position. So we think of standard momentum steps,

we see that by using the path steps. We can take larger steps that

are closer to the correct direction. And if we separate out now just that

momentum turn in our last equation. This is going to be the direction

that it actually takes. And then taking the gradient with

the momentum accounted for as we do with Nesterov Momentum, we have this extra

correction in the right direction. And the Nesterov steps move even more

smoothly towards our optimal value. Now, let's move a bit away from

this concept of momentum and talk about the AdaGrad optimizer, which

is short for adaptive gradient algorithm. The idea here is to scale the update for

each weight separately as we do our gradient descent and

we update our weights. So what will this do? What this will do is it will

frequently updated weights a bit less. And while updating, it will keep a running

sum of each of the prior updates. And then any new updates will be scaled

down by a factor of the previous some so that the steps continuously decrease. So let's look at what this actually means. The key difference when we do

AdaGrad compared to our normal gradient descent is this term G. And this term G will continue to

increase as will be starting at 0 and will keep on adding squares of

that derivative that we see here. And obviously, squares will always be

positive, so G will continuously increase. Then in order to update W,

rather than just using the learning rate, we use the learning rate divided by

the square root of this G value. And since G is continuously increasing, we know that the learning rate

will continuously decrease and this will lead to smaller and

smaller updates at each iteration. So as we get closer and closer to

the optimal value, that learning rate will shrink as we get closer and

will help us avoid that overshooting. Now, I'd like to move on to another

optimization method, namely RMSProp or root mean square propagation

is what that short for. Now, we're working with a very similar

functionality as the outer grad that we just discussed. Except that rather than just using the

some of our prior gradients, were going to be decaying older gradients and giving

more weight to more recent gradients. And this could be similar to the

functionality that we use for momentum. Now, we're just using that weighting that

we discussed for momentum except for the learning rates. And this will allow for updates to be

more adaptive to recent gradients, and is usually much more efficient

than working with just AdaGrad. And then finally, we have this concept of

Adam, this optimizer Adam, which is for adaptive moment estimation. Don't worry too much about

what it's short for, but this will combine both

the concept of momentum and this RMSProp that we just discussed,

putting them both together. So on the left side herem we

had values similar to momentum. If you recall our discussion during

momentum or just going to where you're placing our end with beta one and our

Alpha with one minus beta one which can be used for the momentum in our past

formula as well as we discussed. Now we didn't get into the math of RMS

prop, but I did mention that will work similar to the formula for momentum,

which is what we see here to the left. So to the right forearm sprouts. RBT value which synthesis forearms

proportion is specific to our learning rate. Will have a very similar update to give

most weight to the most recent values. Now, I'd like to note here,

if you're trying to figure out how to default each one of these values,

beta one and beta 2. By defaults, beta 1 will be 0.9 and

beta 2 will be 0.999. And they generally do not need to

be played around with too much, but you can't play with play around with them,

but if you find that you're not

getting to the optimal model. Now, there's going to be a bit of

bias built into each of these terms. So for mt, we're going to want to correct

that bias by dividing by 1- b to the t, and this is meant more for

correction towards the beginning. As you can imagine, as t is growing,

the larger t is the smaller beat of the t will be,

it will continue to shrink as t gross. And then we do the same for Vt,

which again is the RMSProp portion. And finally we update our weights using

our special learning rate, scaled for VT that we just calculated,

multiplied by our momentum term empty. And there we have it or

Adam Optimizer combining both RMSProp and this concept of momentum. Now, which one should we

choose between each one of the optimizers that are available to us? Now, RMSProp and Adam had become

quite popular and from 2012 to 2017 approximately 23% of deep learning papers

submitted to this popular platform for research in deep learning,

mentioned using the Adam approach. Now it can be difficult though

to predict in advance which one of these approaches will work best for

a particular problem, and this is actually still an active area

of inquiry in deep learning research. Now I would say it's important to

note that while Adams speeds up the optimization process tremendously and usually does a fairly good job

at finding optimal solutions, there are going to be times when

it does have trouble conversion. And there are actually even

different versions of Adam that have been implemented in recent,

that discovered recently. With that, I would say whether using

different iterations of Adam or other optimizers that we just discussed

that may speed up the training. If you're still having

trouble with convergence, I would note to at least try using just

regular mini batch gradient descent or full batch or

stochastic gradient descent as well. So, just to recap,

in this section we run over why it's so important to have regularization

with deep learning models. As these complex models are powerful

enough to fit almost exactly to our training data, and with that in mind,

we went over different regularization techniques, such as what we've seen in

rich with adding on a penalization term for higher weights within

that cost function. As well as as we see here in

the next bullet using drop out so that our models aren't overlaying

on particular pathway to the network as well as early stopping,

where we may be checking against a validation set as we trained

to prevent are over fitting. And finally we discussed different

optimizers available to us. Be on that regular gradient descent,

including using momentum RMSProp or combining the two using Adam. Now that closes out this set of videos in

the next set of videos will review some of the extra pieces to keep in mind when

building out our actual neural networks, that will closeout all we need

to know to get started and tuning our own neural networks in Python. All right,

I look forward to seeing you there.

In this section,

we're going to cover some final missing

pieces to keep in mind before starting to

look into actually coding up our own

neural networks. Now let's go over the learning

goals for this section. In this section, we're

going to cover some of the details of training

neural network models. A lot of this will be review as we'll go back over

stochastic gradient descent, as well as other

batching approaches and important terminology. The reason why we do this, is because once we start to actually implement our

neural nets in Python, you'll actually have to tune each one of these

different parameters, that we're going to discuss here. Given our different data

points within our dataset, we now know how to compute the derivative for each

one of our weights, and we went over

different options on how to use that derivative to update our weights using

different optimizers. Now I want to review

how often we should actually go about

updating our weights. As this is going to be something again that we're going to have to tune when creating our

neural net models in Python. What do I mean by how often we need to update our weights? We're going back and reviewing this idea of using

all of our dataset, part of our dataset, or maybe

even just a single row. On our classical approach, we'll be getting the derivative

for the entire dataset, and we'll use that derivative

to update our weights. So using the entire dataset. The pros is that each step

will be informed by all data, but the con will be that this

can tend to be very slow, especially as that

dataset grows very large. Now on the other end

of the spectrum, we again have stochastic

gradient descent. With stochastic gradient descent, we get the derivative

at just a single row, at just a single point and

take a step in that direction. This means that the steps

may be less informed, each one of those

individual steps, but you ultimately

take many more of those steps as you run

through your entire dataset. The hope is, and

the idea being that with us being able to

quickly take more steps, it'll ultimately balance out any missteps you

make along the way. With the idea that you can take missteps at every iteration, you probably want a smaller

step taken each time, so you don't veer too far

away in the wrong direction. Also since it won't be perfectly fitting to the entire dataset, this will also help in slightly regularizing your model as well. Then we have our compromise using mini-batch

gradient descent. Here we'll get the

derivative using just a subset of our dataset and then take a step

in that direction according to the

derivative of that subset. The typical mini batch size will tend to be 16 or 32 rows, and you can tune this

approximately the more rows that you choose, the slower it may take to run. Again, think about the

stochastic gradient descent being a single row

running very quickly. So the larger you have to

run that derivative on, the slower it may take. The idea of this

compromise is meant to strike obviously a balance between the extremes of that full batch gradient descent and stochastic gradient descent. Now just to hammer this all home, let's visualize each of these approaches in

comparison to one another. We see all the way

to the left here, faster and less accurate

steps and all the way to the right we have slower

and more accurate steps. I want you to think, given everything

we just discussed, where stochastic gradient

descent will fall, where mini-batch gradient

descent will fall, and where full batch

gradient descent will fall. All the way here to the left as I hope you

predicted on your own, we're going to have a

stochastic gradient descent, where we'll have faster, less accurate steps and that we see the zigzag going as it

tries to optimize the model. Then on the other

end of the spectrum, we have full batch

gradient descent, which is going to be that slower, but more accurate steps taken. Then finally, we have

our compromise in the mini-batch gradient descent where it falls somewhere

in the middle, it's not quite as

fast as stochastic, but faster than full batch and it's not quite as

accurate as full batch, but it is more accurate than

stochastic gradient descent. Now, just to review some

batching terminology, we have full batch using the entire dataset to compute the gradient

before updating, we have mini-batch which uses a smaller portion of the data, but more than just that single

example that you would use the stochastic

gradient descent and then we stochastic gradient

descent which just uses a single example to compute

the gradient before updating. Though sometimes something to note as you do some

learning on your own, people actually will use

SGD to refer to mini-batch. Be aware of that as

you start to read your own literature in regards to choosing

your batch size. Now another piece of important terminology is going to be this idea of an epoch, and that epoch is going to be one of those hyperparameters

that you're going to have to tune when you're actually implementing your

neural nets in Python. It refers to a single pass through all of the training data. Now what do I mean by that? If we think about a full

batch gradient descent, there will be one step

taken at every epoch because we're setting

how many times are passing through the data. Sorry, I didn't do

every single step we pass through all the data, we do a full epoch. In SGD, in stochastic

gradient descent, there's going to be n

steps taken per epoch. We're going to take as

many steps as there are rows in the dataset every time you run

through an epoch, because again an epoch just means that we have ran

through the whole dataset. Then with minibatch,

there's going to be n, the number of rows divided by the batch size number of

steps taken per an epoch. If you just think

about the dataset being 360 rows and we

say batch size of 36, we will take 10 steps

at every single epoch. When training we often refer to the number of epochs

that are needed for that model to be trained

and that's going to be an important

hyperparameter that we're going to tune as we try to create our own neural

net models in Python. That closes out this video. In the next video, we're going to discuss another

piece of terminology worth understanding, namely

data shuffling.

Now in this video, let's discuss the concept

of data shuffling. So if we think about stochastic gradient descent or mini-batch gradient descent, we'll be going over a subset

of our entire dataset. So to avoid any

cyclical movements, to avoid us going

down the same path as we do our gradient

descent every time, and to aid convergence, it's recommended to shuffle

the data after each epoch. By doing so, the data is

not seen in the same order. If you think about again, Mini-batch gradient descent or stochastic gradient descent so that now you're looking

at the batches in the same order every single time, and the batches are

not going to be the exact same ones

every single time. So let's go over what

this actually looks like. Now if we were to do full

batch gradient descent, then we would run through

the entire dataset, there would be a single epoch, and then there would be

no reordering necessary. Now, if we were to split

this into multiple batches, as we normally would with

Mini-batch gradient descent, for example, there's

going to be a specific ordering that we'd

split it up into with. Batch 1 being a certain subset, batch 2 being a certain

subset, and so on. Then recall that at each

one of these batches, we find the derivative

and use that to move our weights

towards the optimal value. So at each batch we're

taking another step, moving closer and closer

towards this optimal value, and once we run through

the whole dataset, then we've run through

a single epoch. Now, in reality, we're going to run through more

than one epoch. We're going to want to actually have multiple run-throughs

of the dataset, and just to see how many

run-throughs we have here, we split it up into

a bunch of slices. This is meant to represent, even though it's the same length, multiple epochs through

our full dataset. You see that there's not that same ordering of

the different colors. The colors are a bit random here. As after that first epoch, rather than going

back to batch 1, it's going to actually start with some other random batch, and that batch doesn't

even have to be the same batch that

we had before, and that will be the next step, and it will keep running through until we reach that

optimal value. Again, the idea being

that we shuffle around our data so that at each step, we are going to be looking

at a different subset of data so that we don't keep

repeating that same path. Now that closes out this video. Now let's recap what we

learned here in this section. In this section, we discussed the details of

training Neural Network models, specifically working with

different types of batching, such as, we see here, stochastic gradient descent, or mini-batch gradient descent, or full batch gradient descent. With those different

batching approaches, we discussed important

terminology, such as working with epochs and understanding that an epoch is just one run

through the dataset, and depending on whether

you're doing stochastic, mini-batch, or full

batch gradient descent, you'll make a certain amount of steps towards your optimal

value at each epoch. Then we discussed this

idea of shuffling, where if you're going to use mini-batch stochastic

gradient descent, make sure that you're

not just repeating the same steps over and

over again at each epoch. Now that closes out this video in regards to the

fundamentals that we'll need, and in the next video,

we'll actually introduce the library that we'll be using in order to implement our

neural nets in Python. All right, I'll see you there.