We just went through that

notebook introducing Keras. In that notebook, we saw that we needed to

actually implement some transformations

in order to actually ensure that our neural

nets performed optimally. Here we're going to talk about some other important

transformations to keep in mind when training

our neural net models. Let's go over the learning

goals for this section. In this section,

we're going to cover pre-processing and preparing

your data for analysis, so all the steps that are

going to have to come into play when you're thinking about creating a neural network. Part of that will be if you're doing multiclass

classification, how to set it up so that

you can predict across multiple classes rather than

what we've seen so far, where there's just one

class or the other. Then finally, we're going

to discuss the importance of scaling your

neural net models. We saw this in our notebook as we went ahead and we used

the standard scaler in order to scale our data, and you can also use something

like the MinMaxScalar, which we've seen in

the earlier courses. For binary

classification problems, I'm just trying to decide

between two different classes, we have a final layer with just a single node and

a sigmoid activation. We saw that in just

our last notebook where we had a full

dense network, they're all connected

to that final node, there was only one node

in that final layer, and we had a sigmoid

activation function in order to allow

for that output. The sigmoid activation function has many desirable properties. One is that it gives an output

strictly between 0 and 1, and that value can be

interpreted as a probability, so we can say which one is

more likely and by how much. It's going to have

a nice derivative, meaning that it's going

to be easy to find the gradient as well as to use that to do back-propagation. It's going to be analogous

to logistic regression, or you'll have a bunch of

input go into that linearly, go into that node, and then you'll have that one nonlinear transformation

as you do with logistic regression to output that value again between 0 and 1. The question is, is there

a way to extend this to a multi-class

setting if we're trying to predict across

multiple classes? If we want to do this

multi-class classification, we can use what we learned in regards to one hot encoding, and we use this most

frequently when working with different feature variables

and we're just going to use that concept for our

outcome variable. One hot encoding, again,

is four categories, and you can take, for example, a vector with length equal

to the number of categories. Say that your vector

just has one value for each category and those different categories

are going to be, in this case, checking, saving, and mortgage the type of account that

you have there. You can then represent

each category with one at a particular position

and 0 everywhere else. For example, with our

bank account example, rather than just having 1, 2, 3, we can have three new columns where one of those

columns is for checking, perhaps that top

value was checking, we put a 1 there on top. Then that second

value was savings, so we put a 1 in the middle

and zeros everywhere else. Again, that top 0 would

reference to whether that value is going to be

checking that bottom one, or be whether or

not it's mortgage. We put a 1 at that

bottom value because that not bottom

value was mortgage, and zero is everywhere else. For multi-class

classification problems, we're going to let

that final layer be a vector with length equal to the number of possible classes as we just saw on the last slide, and then we can extend the

idea of the sigmoid to multi-class classification

using this softmax function. That softmax function is

just going to be the lead to whatever that z output

was for a particular class, over the sum of e\_z for all

of the classes combined. What that does is it's going

to yield a vector with entries that are going

to be between 0 and 1, normalizing them all

to between 0 and 1, and that will

ultimately sum to one. Then we can get the

probabilities for each one of the

individual classes. For the loss function, when we even inputted it in, it's going to be

categorical cross entropy that we're trying to calculate. This is just going to be the log-loss function in disguise. We take that cross

entropy and that's equal to negative y\_i, y being the actual

values times log of y\_i, whatever that prediction is. The derivative of this will have a nice property when

used within the softmax, so that the derivative of

that last z\_i in regards to that softmax is going to be y\_i the prediction, minus y\_i. It's not like to discuss

actually scaling our inputs. In our discussion of

back-propagation, we briefly touched

on the formula for the gradient used to update

the values of our weights, W. In this, I promise to tie back into scaling our input,

so just hold tight. But note to update our weights, we take the partial derivative

in respect to W and we get again y-hat minus y, which is that first

partial derivative, and the dot product of a, whatever that input was

from the last layer. At each iteration of

gradient descent, W\_new, or new W is going

to be that W\_old minus a learning rate times

this partial derivative. When i equals 0, we are using the

input values of X of those actual inputs as part of the derivative

to update W\_new. Those input values at that first layer are going

to play a large role. This is going to mean

that if we don't normalize the input values, those with higher

values are going to update much more quickly

than those with lower values, because again, we're

using the AI from that prior step in order

to update our values. If we have them on

different scales, higher values will

update quickly and the lower values will

not update as quickly, throwing off the way that we

update our actual models. This imbalance can

greatly slow down the speed at which our

model actually converges. For that reason, we need to scale our inputs in different

ways that we can scale our inputs that

we've discussed in prior courses is the linear scaling to the interval

between 0 and 1, which is going to be

our MinMaxScaling, which is X\_i minus X\_min, over X\_max minus X\_min to ensure they're

all between 0 and 1, or we can do here linear scaling to the interval

between negative 1 and 1, which is just going to be

2 times X\_i minus X\_min, over X\_max minus X\_min, minus 1, and that just ensures

that you have values between negative 1 and 1. Again, we could also use

that standard scaler, sometimes we want these values between 0 and 1 or

between negative 1 and 1, because if you think about using the sigmoid function or the

hyperbolic tangent function, that will allow for each one of our inputs and outputs to stay on that same scale. Let's recap what we learned

here in this section. In this section we discussed pre-processing and

preparing our data for our neural net models. With that, we introduced how we can do multi-class

classification with neural networks using

that one hot encoding as well as the softmax function. Then we discussed the

importance of scaling your neural network

inputs to ensure that you have balanced updates of

each one of your weights. We talked about how

we can use different scalar similar to the

MinMaxScalar or the standard scalar to ensure that each one of your values

is on the same scale. That closes out our

discussion here on different

transformations that are important for your different

neural net models. In the next section, we're going to introduce our first different type of model framework for

our neural networks, namely convolutional

neural networks. All right. I'll see you there.

In this set of videos, we're going to introduce a new neural net architecture called convolutional neural nets. This has been incredibly

powerful for image recognition, and it's now being used to solve many other tasks as well. In order to cover this topic, we're going to discuss convolutional neural networks

in general and a bit about that architecture with

that original motivation of working with

image data in mind. Then we're going to go over many common terms that you're

going to need to know, and we'll understand much deeper as we go

through these videos, such as grid size, padding, pooling, and depth. Now, let's start off

with the motivation behind these convolutional

neural networks. If we imagine an image and the way that an image works

is that each one with different pixels will have a different numerical

value to give you the density within the red, green, blue spectrum,

we can think about it on the

grayscale to start. But the idea is

that there's going to be some type of

relationship between each one of our

different pixels which are going to be on each one

of our different features. The structure of

our neural network, so far, treats all of our

inputs interchangeably. In that the relationship or

that spacial arrangement of those features have

no impact on our model, so there's no relationship between these

individual features. You just have this

ordered set of variables, feature 1, feature 2, and so on, and what we want is to be able to incorporate our domain

knowledge of how images are actually built in building out our

neural net architecture. Now, again, these

convolutional networks that we discuss here were developed

to deal with image data, and the motivation behind

them will become clear as we work through examples

involving image data. But increasingly, as

I mentioned before, these approaches are

also being applied in other common analytical problems of regression and classification, such as working with time series. Now, some thoughts

to keep in mind, when diving into the motivation behind working with

a new architecture, the variables, in this case

the variables are pixels, are going to have a

natural topology. They're going to have

this spatial component that's actually meaningful, and this makes images

different from say, loan default prediction where

the variables do not have this natural topology or relationship in space

from one to another. We'll also want

translation invariance, so when we're trying to identify whether there's a certain

object in the image, we want to ensure that it

doesn't matter the size of that object or the

orientation of the object, it'll be translation invariant. We also want our model to be able to appropriately handle issues of pixel densities changing due

to lighting and contrast. Convolutional neural nets are going to be based on

what we know about the structure of images and also what we know about

the human visual system. The human visual system has receptive fields which

respond to horizontal bars, vertical bars, etc, and

pieces them together. Within data, many of

the pixels which, again, are going to

be our features, or actually tend to have

fairly similar values that perhaps won't add much

information on their own, and we want to keep

that in mind as well. Within these images, we're also going to

want to be able to identify edges and shapes

that exists within that data. Then finally, we'll also want to ensure that it's scale invariant. Again, meaning that it

will classify an object within that picture as a cat no matter the size of that object. Again, this idea of invariance, whether it's the size

or the orientation. Now, fully connected

image networks, thinking about the

number of pixels in an image as that starting

number of features, are all being fully connected to the next hidden

layer would tend to require a vast number

of parameters. Taking advantage of

these structures that we're discussing here will end up meaning

fewer parameters. If we think about this MNIST image that

we're going to see; that's going to be 28 by 28

pixels on the grayscale. That's what we see here. We have this example

of the MNIST image. The idea here is that we have these handwritten digits

ranging from 0-9, and we want to use deep

learning to predict whether that handwritten

image is a 4 here, or 5, 6, 7, etc. Now, this is an MNIST

image on the grayscale. An average color image,

on the other hand, will typically contain

200 by 200 pixels with three different color

channels, red, green, and blue for a total of 120,000 values or 120,000 features to start

out our network. If we imagine with that

fully-connected network, we will have to start

off with at least 120,000 weights just

on that first initial, or 120,001 if you

include the bias term. You can even imagine a

single fully-connected layer would require this incredible amount of weights

if we're talking about that layer being something

more than one or close to what we are talking about the size of that input features. With this many weights, that variance would be

incredibly high with a very high likelihood of

overfitting to your data set. With that in mind, we're going to introduce a bias,

and in this case, a bias in relation to that

fully-connected network, such that the

architecture will be adjusted to look for

certain kinds of patterns. Now, the motivation behind

this new architecture is that different layers can learn

certain intermediate features. We can start off with edges which then build up into shapes, which can then be built into relations between

different shapes even, as well as identifying different

textures with an images. If you think about this build-up, the relationship between

the different pixels will be needed to make the identification of

the slightest edges or these types of textures

that we're talking about. An example of this build-up of features can be understood

by thinking about the identification of a

cat which has features such as two eyes that are certain distance and

angle from one another, as well as having the

texture of cat fur. To identify just an eye

which would have to be a building block to get to two eyes of a certain relation, we would first need

the building blocks of a dark circle, that pupil, inside of another circle or an oval shape since it's an eye, and that circle

will be built from a combination of lower level

features such as edges, and the cat fur should

also be made up of these lower-level edges

in a particular pattern. That closes out our video

providing this idea of that motivation behind that

convolutional neural network. In the next video, we'll

talk about kernels and the actual convolution function that's going to make

this all possible.

In order to capture this relationship between

our different features, those features being the different pixels

within our image. In order to capture

this relationship, we're going to make

use of kernels. Now our kernel's just going

to be a grid of weights, that's going to be overlaid

on a certain portion of our image centered

around a single pixel. Now, once that kernel's overlaid on that

portion of the image, each weight from

the kernel is going to be multiplied by the pixel. Remember that pixel is just

going to be some number, so multiplied by that

number beneath it, and the output over

that centered pixel that we're going to

get by overlaying this kernel on that

portion of the image, is just going to be

the sum of all of those multiplications

of the kernel and its respective pixels. That's going to be the

convolutional operation, and that's where we get this name for convolutional neural nets. This method of using

kernels is going to be what allows us to capture the

relationships of nearby pixels, to detect blurred

portions of images, sharp portions, edges, etc. Let's look at an

example of this in action using a three

by three kernel. If these are going to

be the different values for the pixels in, say our image in this example

is just three by three, and then what we have

here is our kernel. We want to think about how

would we calculate the output? Note that overlaying a

three by three kernel on a three by three image will

only output one single value, and that one single value

will be at the center of what will ultimately

be our output matrix, which we see here to the right. The key will be to overlay that kernel on top of the image. Now, in a way, it's just going to be

like a dot product, where we will take, starting at that first row, cell by cell, we'll take the 3, multiply it by negative 1, that's the top-left corner times the top-left corner

of the kernel, and then 2 plus 2 times

0 plus 1 times 1, and you see we multiplied

each value with its respective point within the kernel across that first row. We keep adding that

up row by row. We look at the second row. We do 1 times negative 2, 2 times 0 plus 3 times 2. Do the same thing

for the third row. Now we have nine different multiplications all

being added up. Each one with their respective

values in the kernel, similar to how we'd work

with a dot product. We add those all together and we end up with this

output value at 2. The way this will work when you're working

with an actual image is your original input will

probably be something larger than three by

three, as we see here. What we do is we just slide over that kernel that we have

using that same kernel, slide it one over to the right, and by sliding it over

one to the right, that will provide the output to the right of that two value

within our output matrix, and similarly, if we had

larger input, again, it's not a three by three input, we get to slide that

kernel one cell down and do all the

multiplication, take that dot products, and we would have the output within our output matrix

right below the two, because we slid it down one. We'd slide that kernel across every single space that it can throughout our input image. Now, you can think of the

kernels as feature detectors. Here, we have a

vertical line detector and there are some

good videos on how you detect an actual line using

matrix similar to what we see here using that

convolutional function. But the basic concept is

just that as you move this filter along some

type of vertical edge, assuming you have

that vertical edge and run that convolution

and get your output, you end up being

able to highlight that there is an existence

of this vertical line. Similarly, we can overlay

the filter that we see here and detect a

horizontal line, or use this filter

that we have here, run it across and detect any corners that we

may have in the image. The point being that

we want to take away from here is that each one of these

different kernels will be able to detect edges, whether they're vertical,

horizontal, diagonal, corners, or other combinations of

features that may be important. Now, these different filters

that we just introduced are powerful to have some type of intuition of what

a filter can be. But in reality, the

network will find those most useful

kernels for you. Also, I'd like to know, we'll probably set up

our framework so that we learn many different

kernels, not just one, but every single one of

these different kernels will operate across

that entire image, and this allows for that

translation invariance. It doesn't matter where

the object is within an image or whether

that object is flipped or what the

size of that object is. Then also compared to our

fully connected architecture. If you think about just having as many different

kernels as we have and each one only

having nine weights, this is going to require much

less parameters to learn. This will reduce that

overall variance in regards to that

bias-variance tradeoff. Now to bring this all home when you're working with images, generally speaking, most of our images will not

just be on the grayscale, but rather have color. For a color image to be

represented numerically, there will have to be,

generally speaking, most common is three

two-dimensional arrays, all stacked one on

top of the other. As we see here, where each one of these two-dimensional

arrays represents either the red scale, the green scale, or the

blue scale, respectively. Now, to move our kernels

to three-dimensions, rather than using the

convolution operation using just this kernel

that's three by three, we're going to use

convolutions on a filter. Filters the term once we

move up to three dimensions, which may be three

by three by three. It's going to be

three, three by three kernels all stack together. So that instead of having nine multiplications added together

to get our one output, we have the sum of

27 multiplications. You could make about the

filters that we learned, where we'll have nine for each one of these

different dimensions, nine for red, nine for green, and nine for blue, and we multiply those

respectively to each one of their different components within that input image to get

our one-centered output. So we're adding together 27

different multiplications. Once we use that filter, we'll go back to

two-dimensional output rather than these

three-dimensions. Now, something that you may

have noticed as we went through this idea of

working with convolutions, is that when we work with these centered values and are trying to output centered values, the edges of our image

and the corners of our image tend to get

somewhat overlooked. In the next video,

we're going to address this problem and introduce

the concept of padding. All right, I'll see you there.

Before we get into

this idea of padding, I do want to discuss a bit about the grid size of our kernels. The grid size is just going

to be a way of specifying the number of pixels that

a kernel sees at once. Typically, we're going to

want to use odd numbers so that there's going to

be some center pixel, but that's not necessary when you move your kernel

across your image, just will be a bit easier to

compute and best practices. Also, the kernel does

not need to be square. Again, that will be

typically what is used, a square kernel, but you do have the options of using

non-square kernels as well. This is our square kernel that we have here with a

height and width of 3. Then if we think about

kernels that aren't square, we have here height

1 and width 3, and we can move this

along our image as well. As well as taking a

height of 3 and a width of 1 and moving this

along our image. Now, I discussed before that as we move a kernel

across our image, it's possible that we do not

put as much weight along each one of the edges and that's going to be

this edge effect. If we use kernels

directly onto our images, the corners and the edges

will not have as much play in allowing us to identify

what that object is. The reason for that

is that pixels near the edge will not be used as center pixels since there are not enough

surrounding pixels. If you think about

having something like a three-by-three kernel and you try to center it

on that top left corner, you won't be able to because you have that three-by-three

and the center of that three-by-three

will ensure that the top and the left of that kernel will not be overlayed on any values within that image. The idea is to pad

and padding adds extra pixels around the

frame of your image so that pixels from the original

image along the edge becomes center pixels as the kernel moves

across that image. Those added pixels are typically

going to be zero valued, so we just call it zero-padding. To think about this

example, first, I want you to look at

our original input, which is going to be

our original image, then I want you to look at

the shape of our kernel. We have the shape of

the original input, the shape of our kernel, now I want you to see

the shape of our output. As we move this kernel along the image and we move it to the right and

we move it down, we won't be able to capture

every single value. The output will actually be smaller than our original input. Also, we won't be able to

center around that one in the top left corner

or the two right next to it to the right

or even the one below it. Anything along those edges, we can't center our kernel on. Now with padding, we add on

the zeros around each one of the edges and we're

able to actually center now on that top-left

corner on that one, and get that output value

that we see to the right. Another thing that I

want you to notice is that we're still using that

three-by-three kernel. But now with padding, because we now have a larger input if we take

into account the padding, our output will be

larger as well and closer to the size of

that original image. Another thing that we can tune when creating our

convolutional neural nets is going to be the stride or the step size as the kernel

moves across the image. We said that it will keep

moving across the image. Normally, if you set

it at its default, it will just move

over one at a time. Along that image, that square

will just move over one to the right then another one to the right until it

gets to the end. Then it will start back

all the way to the left, just one-cell down and then

move along the right again. That's going to be

your step size. You can even set

that to be different for vertical and

horizontal steps. But again, usually, you're going to use the same

value and that will be the defaults and what

you'll see throughout. When that stride

is greater than 1, if we think about the

output that we would get as we do these

convolutional operations, our output if we skip over two rather than just

doing a stride of one, our output will

have to be smaller because we're multiplying or doing less convolutional

operations throughout the rest of our image. So bring down that output value. Here we have an example

of stride equal to 2, so rather than just

moving over one, that kernel moves over two spots, and the next output

would be three. We see our output is smaller

than we even originally had. Our vertical is also going

to be moving down two, so once it got to the

end of the image, it moved down two, we now have our new part of our image that we're

going to run the kernel over and we get our next output, which is just going to be zero. Now, we can combine

this with padding as well and still have

the stride equal to 2. This will be our first

convolutional operation ending up with negative two. We then move over two to

the right and we have our next operation

which will output two and then we can do the

same thing moving down two. You see again that

our output will be just a bit larger as we

add on that extra padding. Now, that closes out our

discussion of padding. In the next video, I want to introduce to

you the idea of adding on depths so that

you can actually pass through multiple

kernels at each one of your different layers.

I'll see you there.

So, as mentioned in

prior videos in images, we often have multiple numbers

associated with each pixel location, thinking about that pixel

location in two dimensions. And these different numbers in that

same location are generally going to be referred to as the numbers for

the different channels. And examples of this include RGB,

which is just the red, green, and blue channels that make up an image. And we saw this a bit earlier

on your computer screen. And then you have a little bit less

commonly CMYK or cyan, magenta, yellow and black for printing images rather than

just displaying them on a screen. Now the number of channels that you have

within your image as they referred to as the depth of that input image. And the filter itself will have

a depth the same size as the number of input channels. So your filter will be as deep

if you're working with RGB, as there are channels, so

there would be a depth of three. So an example of this is if you're

working with a 5 by 5 kernel on an RGB image,

then that kernel worth how many weights? It'll have 5 by 5 by 3 because

it will be four each channel equaling 75 original weights. Now the output from the layer

will also have a depth. So the way that works is the network

typically train many different kernels. Again, each kernel will go over

the entire image, and even though we're working with three dimensions here with

our kernel, when we talked about, for example that 5 by 5 by 3, that's still

going to output a single number. So each kernel outputs a single

number at each pixel location. But you can have many kernels, so

if you had 10 kernels in a layer, that output of that layer will

have a depth equal to 10. And that's because we don't want to

be confined to only working with a single kernel that could

only detect a certain pattern. 10 kernels allow us to detect

10 different patterns. So how is that going to work? So if you look all the way to the left

top left, we have our original image. And in that original image we're

starting off with a 32 by 32 by 3 image. And that's going to be the data

from that original image. We have the three there in that relates

to the red, green and blue dimensions. And then each one will be 32 by 32 for

the red, for the green and for the blue for each individual channel. And then in the next layer we see

that we have a 32 by 32 by 10 layer. And that means that our

depth is equal to 10. So how do we get that depth equal to 10? The 32 by 32 each one of those 32 by

32s will represent a single kernel. So we see that we have that

kernel that's 5 by 5 by 3. If we were to take one section

an run that convolution operation, then we get that single data point,

that's 1 by 1 by 1. And we can do that by moving

that 5 by 5 by 3 kernel along the entire image to get the next

output to get that pink slice that you see within that 3 dimensional

cube in that second layer. So that's how we get one

single layer out of the 10. And since there's 10 different filters,

if we look down to the image in the bottom row,

we have another filter in green. And that green filter moves along

our image an produces another one of those 10 layers. And each one of our different filters

will produce a different layer. Now I do want to note that if you

are using a 5 by 5 by 3 filter and that's moving along the 32 by 32 image,

then you probably need some extra padding so that your

next layer will still be 32 by 32. And you also have to take minimal

strides as you actually go about moving from 32 by 32 to another

layer that's also 32 by 32. But the idea is that the number of filters

you have will be the depth of the next dimension of the next layer. So now I want to introduce another concept

that's important in convolutional neural nets, and that's the idea of pooling. And pulling will reduce the image size

by mapping a patch of pixels to a single value. So that will shrink

the dimensions of the image. And it's not going to need any parameters. Though there are different types

of pooliong operations, but every single one of those different pooling

operations will be something like a max or an average where you just going to

take whatever values are output and take the maximum value or

the average value, whatever it is. So speaking of the different types

of pulling, we have max pooling. And with max pooling for

each one of our distinct patches, that pooling will represent

the maximum for that patch. So an example,

here's were using 2 by 2 maxpool and we take our original image that's 4 by 4. And we split it up into each

one of these 2 by 2 squares and we get the max value within each one

of those square to reduce the size of that data set to that 8 1 5

4 that we see on the right. And then the average

pool is self explanatory. It's in the name, whereas we take the each

distinct patch and we get the average. And we can see again how we perform

similar to what we just did before but rather than taking that max value,

we take the average value. I would say taking the max value is

generally much more common practice in regards to what you'll be using when

you actually pull together your data. So, just to recap., in this section we gave you an idea of

what convolutional neural networks are. What that convolutional operation was. And how we can use things such as

the filters in the kernels in order to come up with our next

layers within our network. We discuss the original motivation and

why we would want to have a certain type of framework one we are working

with image data specifically. And how we can even use RGB using

an image with three dimensions where one of those dimensions is

the number of channels in order to actually come up with our next layer. And with that we discussed

things such as the grid size and how we'd use that grid size to move

along our image, adding on padding so that we wouldn't have to lose

information along those edges. The idea of pooling to reduce

the number of dimensions, whether that's max pooling or average

pooling as well as this idea of depth, where each one of your different filters

will add to the depth of the next layer. Now that closes out our discussion

here on convolutional neural nets. And in the next video we

are going to have a notebook where will actually see convolutional

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

LAB

Welcome to our notebook here on Convolutional Neural Nets. Here we're going to be using Python to build out our

Convolutional Neural Nets in order to classify images using this famous CIFAR-10 data set. This CIFAR-10 data set is

going to be 60,000 different images each 132 by 32 pixels

and their color images. They will also have a

certain amount of depth, if you recall from the lecture. Each one of these

different images will be one of 10 classes, either airplane, automobile,

bird, and so on. Now, in order to build out

our Convolutional Nets, we're going to have to

introduce new parts of Keras, new functions, and

new layers that we hadn't used in

our prior notebook. We introduced the

sequential model, we can still use that. At some point we will have to use that dense connected layer, similar to what we did with

that fully connected layer. We can also use

dropout in order to regularize and ensure

that it doesn't over fit. We'll have our different

activation layers that we can use as well. Whether it's ReLu or sigmoid or hyperbolic, whatever it is. Then we also have

this flattened layer. This flattened layer will be

important as we move from our convolutional layers

to our dense layers. Eventually, in order to make

some type of prediction, we're going to need to

flatten it out and then have that dense layer connected

to that final prediction. We're also going to import this Conv2D and

this MaxPooling2D, which will allow us to build out our convolutional

layers as well as our pooling layers that

we introduced in lecture. Now, to get our data, we imported earlier this CIFAR-10 from the Keras data set, so this is actually

within the Keras library. We have this data set available. We call load data. When we load data, that'll

give us two tuples, our training set, as well as our test set and the x

and y values for each. Now we're going to print out the shape of our training set, as well as the

number of samples of both our training and

test set, separately. There's going to be 50,000 train samples and

10,000 test samples. We will be training on those 50,000 train samples and then we can ultimately tests on

that whole set on 10,000, but what I wanted to look

to is the shape of x\_train. If you think back to what we've

been working with so far, and those were not image data

that we were working with, here, you see that we

have four-dimensions. The first dimension is going

to be the number of rows or the numbers of samples

that we're working with, and that's 50,000. Then the next one is going

to be the height and width in terms of the number of pixels of our images

as well as our depth. That's why we have the 32 by 32 and then by 3 for the red, green, and blue,

different layers. We can look for each

individual image that we have this 32 by 32 by 3 shape. That's going to be a

bunch of numbers from 0-255 for each one of

these different colors, red, green, and blue. We can see that these actually

represent actual images. We see here that this is of class nine if we call y\_

train and just 444, then we can look at the x\_train, which is going to

be the actual image itself without the label. We see that it's actually

going to be a real image. It's not a high-definition image. If we get 32 by 32 pixels it will not be a

high-definition image, but we can tell you

that we have a truck and hopefully our

convolutional neural net, will be able to pick up the

fact that it's going to be tires and the large back and whatever other features there are that build out that truck. If we look at our

y\_train originally, we see that it's just

a bunch of numbers, each one representing

a different category. As we discussed in lecture, we often need to take something

that's categorical in many different

categories and turn that into a categorical variable

doing one-hot encoding. Keras has functionality

built in to change this output into that one-hot encoded

version of the output. In order to do so, we call Keras.utils.to

categorical and we say what the set is that

we want to change to categorical and the number

of classes in that set, which is going to be equal to 10. We run that. Now if we look at that y\_train

that we had above, which we see was nine, the new value is going to be

that one-hot encoded version with zeros everywhere

except in the nine spot. Then we're going to want

to make sure all of our values flow and scale

down to between zero and one. Recall that all of our

different pixels will be values between zero and 255. If we divide by 255, we ensure that all of our values are going to

be between zero and one. Now, when we use this

convolutional neural nets, when we create these layers, these convolutional layers we call just same as

we did with dense. We call Conv2D and we want to ensure that we understand the different

parameters that we can pass through so

that we can specify exactly what convolutional

layer we want to use. Thinking back to lecture, some of the important

parameters that you should know is going to be the filters, and that's going to

be the number of filters used per location. In other words, that's

going to be the depth of your output or the

number of kernels used. If you think about

again, that depth, we are call in lecture, we had at one point a depth of 10 and that was because we

had 10 different kernels that will output a depth of if we said 10 of 10 if we

set filter is equal to 10, then we have our kernel size, which should be a tuple giving the height and

width of the kernel used and you can specify the height and

width to be different. If you just pass

through one number, it will assume a square, and I would say stick

with squares to start, you can try playing

around with other values, but those are going to be best practice for the majority

of your starter material. Then we have the strides. That's going to be how you

move along those kernels, along your image and whether you want to

move it one at a time, going from left to right or two at a time going

left to right, as well as up and down. The first value is

going to be the stride going left to right and then

the next one is going to be up and down and then you're going to want your input shape, which you will recall we

pass through when we had our dense neural network and

that was just one value. Here, if you just recall what we pulled out in

terms of the shape of a single image that's

going to actually be three-dimensions and we want to ensure that it fits

within that first layer that we specify to ensure

that that's correct. One more thing that

I want to point out that's not here is the padding and we set

padding equal to valid. That means that we are not

having any padding and it'll stop as soon as the right if we're moving

from left to right. As soon as the rightmost part of our kernel hits the edge as

we move along those strides. If we imagine that we have

a six-by-six image and our kernel is five-by-five, then it'll only move to

the right once and then stop and then if we

set padding equal to, and we'll see this

later on saying, then that will pad

on some extra zeros. Generally speaking, to make sure it'll be just one

set of zeros around, but maybe it's even and

out there might be two on one side and there'll

be padding with zeros. Then again we have this

flattened layer and that turns our whatever input has into a one-dimensional

vector and that will allow us, once we do that, to try and just

transition between the convolutional layers and those fully-connected layers. We have here the

initialization of our model using the

sequential function. We're going to be using that sequential API again in order to build out our model. We then add on our first

Conv2D layer and this isn't the ordering

that we saw above and we can see if we

just call Shift tab, it's got a couple more times. We can see that we're setting

the number of filters so that depth also are

going to be equal to 32. Our kernel size is going

to be five-by-five. We're not going to use

the default for strides, we're going to set

that to two-by-two. We're also going

to add on padding. There's actually going

to be padding on this layer rather than

defaults of leaving it as is and then we specify the input

shape and if you recall, the shape of our x

train is going to be the number of different

samples we have, then the actual shape and

if we say one through, then we're just specifying

the shape of a single object. We added on that

convolutional layer. We then have our activation

which is going to be relieved to ensure that

we have that non-linearity. We then add on another

convolutional layer. Again, setting the number

of filters equal to 32. We are going five-by-five in regards to our kernel

and recall that if we do five-by-five then we're

moving along our image and that will continuously reduce the size of each one of the

layers as we move across, especially since our

strides are two-by-two. We then add on another

activation layer. We can then do our max pooling, which will just take the max of a certain grid and we're setting that pool side equal

to two-by-two. We're going to reduce very

quickly the size of our layer. We're then also

going to introduce some dropout to add a

bit of regularization. We then flatten that out so

that now we're working with just a one-dimensional

object rather than that three-dimensional

object that we're working with in the earlier layer. We can then add on a dense layer so that we have one fully connected layer again, call relu, so we have

a non-linearity, again called dropout

for some extra regularization.Then

we're going to add on that final dense layer so that our output is equal to the number of classes we have. Because ultimately, if you think about

our neural network, it needs to specify, it needs to predict one

of these 10 classes. Then we set our activation

equal to softmax as we do when we're trying to predict

amongst multiple categories. We can call them model\_1.summary. We can see here the number

of parameters at each step. We can see that our output shape is going to reduce at each one of these steps or reduced at

first to 16 to 16 by 32. We kept the depth that

32 at both steps. Then we have six by six by 32. We call the max\_pooling

and that reduce it to a three by three by 32. Then we had our dense layer and you see there's a turn

of parameters there. Recall that these

dense layers are going to have a lot

more parameters and be a lot more variance than we would have with

our convolutional layers. We have one more dense layer and then our final activation. We can then specify our batch size as well as the optimizers

we're going to use, so specifying we

want the RMSprop, optimizer with this

learning rate. We can specify the decay if

you recall from RMSprop. Then with that, we compile using categorical\_crossentropy rather than our

binary\_crossentropy. We can specify the optimizer

metrics we want to track and then actually fit our model with

the batch size, specifying the number of epochs, we have our validation data to see on the holdout

set how it does. We can shuffle equals

true and that'll just be in regards to as we optimize, we want to shuffle

our data throughout. I'm going to let this run and we'll come back once

it's done running. I want to warn you this

may take some time. That should have taken

maybe five minutes, maybe a bit longer

in order to run. Am looking actually

to see the timing for each epoch as we run through. We have here the 15

different epochs. We can see that the loss

on the training set continuously went down

for each 115 epochs. We didn't save it as

we did before so that we can access that history

key, that history dictionary. But we can see what

happened at each step. We can see that we're tracking that validation loss and that goes down for the first

number of epochs. Then round here we

see that it starts to fluctuate or it

goes down, back up, down again on that

validation set looking from 1.049 - 1.08. Then we can see that the accuracy rather than continue going up, starts to fluctuate as well

on that validation set, but it continues to increase

for that training set. Now, if we wanted to do

any type of prediction, we can do the same

thing that we did in our last notebook by taking that model and

calling if we want, the probabilities, we

can just do.predicts. We can call that on our x \_tests. We run that and that'll give the probabilities for each

one of the different classes. If we want to predict

the specific class, we can just

call.predict\_classes as we did before and we see here that has a prediction

for each class. Now if we recall, if we want to test our accuracy, let's say, or any other metric or I'd say, if we want to look

at the accuracy or something else that requires

that actual prediction. If you recall, our y\_tests

had been converted to this one-hot encoded

version of y\_test. We'll have to take the inverse ticket back to

what it was originally. In order to do that, we can just pull a numpy

and call np.argmax. That'll just say, it was

the maximum arguments. You have to specify

across axis-1. When we call that, we can get each one

of the actual values. We can see from what

we have here probably predicted correctly for 3, 8, 8 and 5, 1, 7. The accuracy score should

be what we have here, but we can actually test this. We can import from

sklearn.metrics, accuracy\_score. Then we can take

that accuracy\_score of what we have here. These are the actual values. Then our prediction

that we have here. We have that same value, 0.6176, since that was

used as a validation set. That closes out that

first exercise. In the next exercise we will

walk through building out a different convolutional neural net and see if we can make any improvements on

our current model. I'll see you there.

Welcome back for our next exercise. Now in our previous model,

we had the structure that we have here, convolution layer, another convolution layer that Max pool to

bring down the size, flattening it out. Then that dense connection, and then that

final classification with the activation functions in the dropouts

that we had specified. Now we want to try building

a more complicated model, and it's going to have the following

structure, convolutional layer, convolution layer, Max pool, and

then two more convolution layers. So we're adding on an extra two

convolution layers, another Max pool, and then that flatten that dense

connection in our final classification. We're also going to use rides of one for

each one of our convolutional layers. So rather than moving that kernel along, two to the right,

then two down as we did before. We're only going to move it across,

and down by one each time. We're then going to see how many

parameters as our new model have, and we compare to our old model, and then we're

going to train it only for five E pox. It will be more complicated, so it will

take some more time, and then we can look at the loss in accuracy numbers for

both the training and validation sets. And we can, on your own, go ahead and

try different structures and runtimes and see how accurate you can

get your model to be. So we going to run this with

this specified new framework, so we're going to again,

have 32 different filters. Here our grid is going to be 3 by 3. Above if you recall, we had 5 by 5, so

that will also move across a bit quicker. And then before we had destroyed

the strides equal to 2 by 2, now they're going to be at

their default of 1 by 1. And then we're having padding on each

we'll also add on some extra weight, some extra learning that we'll have to do,

it'll have to go through more convolutional operations

as we have that padding. Then we have a relu activation,

and again we set the default, we have another convolution layer,

this time without padding. We have another activation of relu,

some Max pooling, and then we have another convolution layer,

this time with 64 different filters. And we're going to do that with padding,

and then again without padding

again using a 3 by 3 grid. And then will flatten,

have are dense layer and then our final dense layer to predict the classes as

well as the activation of Softmax. So we run this to set up

our new framework, and then when we look down at the number of

total parameters that we have to learn, or up to 1.25 million total

parameters that we have to train. And if you recall, before we only

had around 181 thousand train. So if we think about the timing that this

will take, and we'll start to run it here. We're going to have probably something

that's going to take a lot longer at each one of E Pox, so we see that

ETA is going down pretty quickly. But still, each one of our E pox, it's around this three minute mark

that's getting three minutes 20 seconds. And it's going to take some time

at each E pox, compared to what we had before what was going through

each E pox, around 27 seconds. Now I'm going to pause the video here,

and come back when this is done running, and this will take some time to run

even longer than it did before. But it's something that we want to

make sure that you take into account. As you start to build out your

deep neural networks, and understanding that as you

have more complex structure. You'll probably need a stronger machine,

or some way of paralyzing across multiple

machines as you build these out. Alright, I'll see you in just a bit. So hopefully you able to

run that on your own. And as we see here,

it took quite a bit of time to run that. We see that's a bit under three minutes

for each one of the different E pox, for five E pox we're getting close to 15

minutes to run through and fit the model. But what we also see,

is if we look at the accuracy, and specifically be validation accuracy for

that hold out set. We, after the fourth E pox got to

a higher accuracy than we ever got before with the other architecture. So we see this more complex framework was

able to better fit to our actual data set. Now we can play around with different

frameworks, adding an extra convolutional layers, or moving convolutional layers,

changing the stride and so on. But as we saw here,

we could take some time. So and because of the flexibility,

there's actually some architecture. Some frameworks that are best practices,

or most common practices that are used throughout that

will discuss in just a bit. But before that, in our next video will

discuss how we can use something that we trained on for a specific data set,

such as what we did here. And use that training

to actually supplement. A classification of images for

completely different data set, and we'll see what we mean in

just a second when we discuss in the next lecture the idea

of transfer learning. Alright, I'll see you there.