In this set of videos were

going to motivate and gain an understanding of how

Recurrent neural network work. Now let's go over the learning goals for

the set of videos, in this set of videos, we're going to cover what

Recurrent Neural Networks are, as well as the motivation behind them. We'll discuss both the practical and

mathematical details, that allow you to understand how

recurrent neural networks work. And then finally, will touch on some limitations of these

recurrent neural networks that we discuss, that will lead into our next set of videos

on how to adjust for those limitations. So we discuss how processing of images, will force them into

a specific input dimension, where with our grayscale we can imagine

the two dimensions of, pixels a 28 by 28. And why something like a convolutional

operation, which we saw in the past videos takes on surrounding cells, and it makes

sense for these types of input data. But this may not be immediately obvious,

in regards to text, in regards to what kind of data we want to input on,

what kind of operations we want to use. For example, if our problem statement

was to classify tweets as positive, negative, or neutral. Different tweets can have

different number of words. And we want to know, how can we

account for this variable length for each one of our input sequences. Now we want to do better, than just

the bag of words implementations, which would essentially take every word,

and just state how many times that

word appeared in the document. Ideally, one working with text data,

each word can be processed and understood in the appropriate context,

and by context we can think of it as the prior word surrounding that word,

prior sentences, etc. And those words should be handled

differently depending on that context. You can think about, say,

about being either the animal of a bat, or a baseball bat. Also, we get more words,

as we get those more words, we should be able to update the context

that we are currently working with. So the solution will be, to use this

idea of recurrence where we input the words into our network one by one. And this would mean that, we can deal with

variable lengths by just continuing to feed until the end of a sentence,

or till the end of the document. And because we have

the information from prior words, the response to any particular word, can

depend on those that actually preceded it. Since we're feeding it one by one. Our network would then, output two

things as each new word came in. One being a prediction, if the sequence

were to end at any particular word, what would the prediction be? And second a state which

contained a summary, of everything that happened in

the past leading up to that point. Or again,

that context that we're looking for. Now this picture of how

the recurrent network is used, or how it's built is often a bit confusing to

digest, but what we're looking at here. Is that the input will come

into our network, one word or one time step at a time. And as those values come in, we can

update the state with all past contexts, so we can keep track of

the inputs that come in. As well as out putting a value, so that

we can have a prediction at each word or at each time step. Now, as I said,

this may be a bit unclear, so let's unroll this recurrent neural net and

take a deeper dive inside. Now looking at the quote Unquote,

unrolled version of what we just saw. We're going to have our

words coming in as input, and those come in one at a time,

and then starting back at W1, we have a linear transformation

denoted here by this matrix U. And note that, W will be some vector

representing that single word, and in general,

RNN does not only take in words, but can be taking in any information

at one point in time. So you can imagine this

being an input vector, with sales data with inventory levels. Promotional spends all for

one time period and then W2 being that same information for

time period two, and so on. But again, in this instance we're just

going to continue to think of this in terms of words, where each word has its

vector representation for word one or two, etc within the sentence. Now how are we able to store, and pass that information from

one cell to the next? The way that we do that,

is that each step, along with our input dot product of W and

U, which we just did right? We took the dot product of that W vector,

with our U transformation, we're also going to be getting as

input the state from the prior cells. So starting at S1, or even before S1, we can initialize that state

with a zero vector, but then we pass that information from S1,

or State one to S2, and so on. The way that we do this, is we add

together the values from that prior state. And take the dot product of

that state with our matrix W. We then combine the values

from the input of W 1 and U. As well as S1 and that W matrix,

and pass that through altogether, all combined through an activation

function to get our new state. Now the output from this activation, can be used as actual output at each step,

and that is often the case. We can also do as we see

here with these V matrices, is have another transformation

take place with this vector. Pass that through another activation, and either pass that new value as output,

or even create another layer on top. You can think of this as just

the first layer in our neural network, with some amount of nodes. We can have it another depth to our layer,

and create another layer taking in

as input these output values. And once all that is done, often we

may only use that final output that we have here, to create our ultimate

prediction that we're trying to make. So as an example. The first two words have

an unknown sentiment, or the last two words that we have here, are

going to have that positive sentiments. So you see the question mark, and then it

was able to predict positive sentiment, if it's predicting sentiment at each

one of these different outputs. Now each of these cells can have

an output greater than one, so we can imagine if we set our matrix to

have an output with, say five values, so that O1 is an array with

five different values. We would be getting five different

outputs at each one of these steps. And this is the idea of

having more than one node, in our first layer within a fee for

a neural network. Those are one in the same. So if we are assuming something like five,

or more nodes, or five or more outputs, and

ultimately we want to predict something, like a class that's only between

two values or three values. we'll need to have a dense matrix, that

give the linear combination of each one of those nodes, as well as an activation

function, that results in either values, or only a single value depending

what we're looking at. Now, if this is a bit confusing,

an important note, is that usually were only

looking at that final output. So here say O4 output four. And since that is the only output with

information from all the other inputs, that's going to be the most important. And that single output of O4, can have

those five values that we just talked about, or 32 values whatever

amount of values you want, in regards to the number of nodes

you want in that first layer. And if we have something like five or 32

nodes, then we need to pass that through a dense layer, just that O4 in order

to come up with the prediction. Whether that's an output of

just one value or three values, whatever it is you're trying to classify. Now what we have here, what we have circled here is really

the crux of our recurrent neural net, which passes through that save state from

all the prior inputs within our sequence. In Keras we call this part,

that we have as the input, the kernel and the kernel, refers to the matrices

used for that input transformations, those use and

we can initialize these weights, using our kernel initializer and

will see that in the notebook. And then we also have weights, within

our current portion of our network, and that will also need to be initialized, and those are going to be

the Ws that we see here. Now that closed out this video,

and the next video, we'll start to walk

through at a high level. The actual math of how this all

works alright, I'll see you there.

In this video, let's get into the mathematical

details behind recurrent neural nets. So starting out, we have our inputs wi, where I represents the ith

position in our sequence, and as we talked about so far, that would

be the ith word within our sentence. And with that we also have si,

which is the state at position i, that holds all pass information that

should be passed through our network. We have oi,

which is the output at position i, and to calculate si as mentioned

in the previous video, we take a function of the linear

combination of our input. Add it on to a linear

combination of our prior state. And that function should be some

nonlinear activation function. And then to get our output or our final

output, we take a linear combination of our current state, and

pass that through the activation function. And assuming we're trying to predict

classes that maybe a softmax for example. So what are we doing here? We get our current state as a function

of the old state, an that current input. We then get our current input as

a function of that current state. And we learn the appropriate weights for these function by training

through our network. So what are going to be the different

weights that we need in order to get that current state? Now, if we think through

our matrix multiplication, thinking through all of this is

passing in just a single input. In reality,

we would pass through a batch at a time. Then we're starting off with

an input of dimension r, and in our example that represented

a single word from our input, so that's going to be

a vector of dimension r. We then have s is going to be

a dimension of our hidden state. And t, we're going to use as

the dimension of our output vector, after passing that through our

dense layer at that final layer. So in order to get

the transformations that we need, and thinking back to that

visualization that we saw earlier. U will be s by r matrix, so

that will take our r dimensional vector or word vector input, and

return something that is an s vector. So it's the same shape as our state. W is then and s by s vector, so

we take that prior state of dimension s, and keep it in dimension s so

it will still be an s vector. And then finally V is going to be a t

by s vector or a t by s matrix, and I will transform our s vector from

that hidden state, into something that is of size t or t vector that

fits the dimensions of our output. And with that we should note that the

learned weights U, V, and W are going to be the same across all positions, so

we saw that unrolled version of RNN. And we had that U show up repeatedly. We should note that not U, or that V,

or that W will be the same throughout. And as mentioned as well. Throughout we will often ignore

the intermediate outputs, and only care about that final output that

has seen all inputs from our sequence. So think about that unrolled RNN,

we discussed that output for being that final output. We only really care

about that final output. In order to train recurrent neural nets,

there's going to be a slight variation to our normal back propagation method,

called Backpropagation Through Time, that allows us to update the weights

within our recurrent neural network. Now we're not going to get into

too much detail about this. One can imagine that recurrent

neural nets must learn weights by updating across the entire sequence. And thus if the sequence is very long,

we are even more prone to that banishing exploding gradient problem than we are

with our regular feed-forward neural nets. And in practice, we're going to set

a maximum link to our sequences to ensure that they don't get too long,

and with that in mind, if the input is shorter than that maximum,

then we just pad that sequence. And if it's too long,

then we would truncate it, and this ensures uniform input lengths for

all of our sequences. Now we touched on this briefly earlier. But although RNNS are often used for

text applications and those with examples we've seen, there are multiple

uses for working with such a framework. They can be used for all types of

sequential data, including customer sales, loos rates, or network traffic overtime. Speech recognition, so working with audio input for call center

automation, and voice applications. For manufacturing sensor data, to tell where along a chain

failure may happen to occur. And as even been extremely powerful

in regards to our ability to now do genome sequencing. So we talked all about RNNS and

some of the powers of RNNS. But one of the major weaknesses of RNNs is

that the nature of that state transition as it's currently constructed,

makes it hard to leverage information from the distant pants, or

in other words, early on in our sequence. With that in mind, in our next lecture

we're going to introduce LSTMS or long short-term memory, which you similar

concepts to what we just learned. But I have a more complex mechanism for

updating the state that allows for longer term memory. So that closes out our section

here on recurrent neural nets. In this section we discussed

recurrent neural networks, and then motivation in regards to learning

neural networks for sequences. We discuss the practical and mathematical

details, for how it allows for providing context for our sequential data. And then finally we touched on the

limitations of recurrent neural nets, and accounting for information

throughout the entire sequence, especially those longer sequences. And how in the next video we're going

to introduce LSTMS to help account for such issues. All right, I'll see you there.

Welcome to our demo here

on recurrent neural nets. In this demo we're going to be using recurrent neural nets to classify the sentiments

on an IMDB dataset. IMDB is just going to be

movie reviews in general. Our data consists of

25,000 training sequences, so those are just going to be sequences representing

different reviews. We're also going to have

with at 25,000 tests sequences that we can test how well we're able to

train our data. Then our outcome will be binary, either it's a positive

review or a negative review, and we have those will

be labeled for us. Keras provides a

convenient interface to load in this data, and this is actually built

into Keras as it's dataset. We'll immediately encode

those words as integers, and those are going to be based

on the most common words, and we'll see in just a bit how those are encoded as integers. Then from there,

what we're going to do is we're going

to actually show you how to come up with

vector representations of those words and then train our actual recurrent neural nets. First things first, we import all the

necessary libraries. So Keras, some that I want to point out here is that we're going to bring in embedding, and we didn't talk

about this much. But when we use embedding, what we're doing is

we're taking our, in this case the integers, but taking those

sequences and taking those words and coming

up with the word vectors that will represent the syntax or the context

of that word in a way. If you have two words that

are basically synonyms, such as doing something fast or doing something quickly

or fastened quickly, could have very similar meanings. The embedding will have vectors that are very

similar to one another, so that gives you another

layer of learning that we're going to come up with, and that's going to be

our embedding layer. We're also going to

import simple RNN, so just going to be that

recurrent neural net. We talked about how that

is going to be simpler than the versions we're

going to learn in later videos such as LSTMs, and that it may have

that problem of that longer-term learning

within a longer sequence. But just to learn how the

cells piece together, when I start off here

with a simple RNN. We're then going to initialize the length of our features, and we see that max

features is 20,000, and this is used when

we're loading in our data. Using the IMDB dataset, it's going to pick the most

common whatever number of max features we

have here 20,000, the most common 20,000 words. We're then going to,

and we discussed this in lecture a bit, set the maximum length of the sequence and will

truncate after this, as well as pad if it's

not up to that length, and then we'll decide our

batch size here as well. When we do our deep learning, of course we always say

what the batch size is, so when we went

through one epoch, that'll be decided by whatever our batch sizes and the size of the

entire dataset. So you can imagine

with a batch size of 32 and quite a large dataset, we'll go through

many iterations of gradient descent before going

through a single epoch. We're then going to

load in our data, and when we load in our data, we just call an IMDB.loaddata, and it's the only parameter

we needed to pass here, or that we did pass

here is going to be that max features

of there 20,000, which again is the 20,000 most common words

within our dataset. That will output x train or y train as well as x

tests and a y test. We can look at the

length of each, and they should be equal to that 25,000 that we just discussed. We'll take just a

second to load here, and here we see 25,000 train sequences and 25,000

test sequences. Now, as I mentioned, we're also going to pad

or truncate each of our sequences using

that max length that we discussed earlier, which was equal to 30, as we see right over here. So we set sequence, pulling in that sequence

that we pulled here, in terms of, from that

pre-processing library, we have the functionality

of padding our sequences. So Keras has something built in, in order to quickly pad or

truncate our sequences, and we set x train

to that max length, as well as our x tests

to that max length. Now when we look at the

shape of those sequences, we see that they're now at

25,000 different examples, where each one of those

examples is of length 30. If we want to see what one

of those examples look like, we see here that, again, this is meant to represent

a bunch of words, and each one of those words are represented by a single integer. Now, our goal is to build out

our recurrent neural net, and in order to do so, we should dive-in a bit into what this

embedding layer is, as well as how this

simpleRNN layer works. So rather than using

pretrained word vectors, we're going to learn

what those vectors actually are using

this embedding layer. Now, when I say that, again, that embedding will allow

you to have that context, so that similar words will have vectors

close to each other. So if we're talking about

X dimensional space, let's see, what was our

dimensional space here? We put in word embedding of 50, we see it down here, then we are going to have a

vector that has 50 numbers, and in 50 dimensional space, one vector should be close to the other if they're

similar in meaning. We're going to learn

whether they're similar in meaning using this

embedding layer. The layer maps each

integer into a distinct, dense word vector of length, output them, as we

just mentioned, and we can think of

this as learning that word vector

embedding on the fly, so using the context

of IMDb reviews, So we'll be specific to IMDb, which could be powerful. Something to note, if you're trying to do embeddings

on your own, there are pretrained embeddings available, such as Word2vec, and because that's pretrained, it makes it easy to actually

take whatever dataset, and automatically use that

embedding and come up with vectors that are similar to one another if

they're synonyms. We then are going to have, again, that input dimension should be the size of our vocabulary, and then the input

length specifies the length of the sequences that the network's

going to expect, and we just discussed how

we're going to keep that at 30 by padding or

truncating accordingly. We then have our simpleRNN layer, which we're going to pass

in the number of units, so we can think again to

our diagram that we saw earlier and we can say how many units we

want that to output. We can say what type

of activation to use, tanh is usually best as we

pass through our simpleRNN, but we have our options

of working with others, and feel free to play

around with that. Then we have our

kernel initializer and our recurrent initializer, which are going to be

the initial values for our weight [inaudible]. Again, that kernel initializer is going to be the

weights for the input, and the recurrent

initializer is going to be an initialized weights

for those state layers. Here, we're actually

going to change that activation to

relu, if you see. You can try going back to

tanh and see how that works, and then we're also going to just pass in that input shape, which is just going to be, if we call x\_train.shape1:

then we should have, and we can just look

at what that is, x\_train.shape1:, and we see that that's

going to be of shape 30. It doesn't matter how

many examples we have, in general, when you're

trying to pass a shape, you're going to be passing in that shape of what a single

vector would look like. Let's build out our first RNN. Our rnn\_hidden\_dim is

going to be equal to five, our word embedding dimension

is going to be 50. So again, we're going to take those integers that

we currently have, and given their context, come up with an embedding, where it's going to transfer each one of those single values into a vector that's

of dimension 50. We're then going to

initialize our model, add on our embedding layer, pass in the max features, as well as the word

embedding dim. That max features is going to be what we have here, 20,000, to give us what the actual

input dimension is. Then our word embedding is going to tell us our new dimensions, and then that's going

to be the first layer. Once we have our new embedding and our data ready

to be fed forward, we can pass that through our simple recurrent

neural network. We pass them in the number

of hidden dimensions, which is just five. We then call our

kernel initializer as well as our

recurrent initializer, again initializing

those weights for that first layer for our input as well as

that state layer. What this is is just

random normal with very tight standard

deviation around that zero for random values, and then this is just going

to be a diagonal matrix where along a diagonal we're going

to have a bunch of ones. This shouldn't make that large of a difference

starting off, you can try just

removing these and using the default values

which we have up here. I've tried it and I believe they're around

similar performance, I think this out performs

it by just a bit. We then set our

activation to ReLU, input shape equal to that

x\_train.shape that we just saw, and then finally to

get just one output, because we just want

positive or negative, we add on that dense layer with the activation of sigmoid. Now we have our model and

we can look at the summary, and we see we have

to train a bunch of parameters for that

embedding layer. Then for the simple RNN,

if we think about it, we're going to have in

that initial matrix, going from our input

to our state layer, we should have a 50. We have 50 as input, and then we have

five hidden cells. We add on that bias term, so we end up with 50 times

5 plus the five bias term, so 255 weights there, and then to go from one

state layer to the next, we recall that we're

going to use a five by five matrix that keep

that same dimension. That's going to be another

25 weights that we learn, and that's how we

get 280 parameters that we are currently learning, then finally that dense

layer which will just be those five input

plus the bias term. We can then call our optimizer, we're going to use RMSprop with a learning rate of 0.0001. We're going to use

binary cross entropy since we're deciding

between 0 and 1, we're going to use that optimizer

that we just discussed, and we're going to track

that accuracy as well. Finally in order

to fit that model, we can pass that in, passing our x\_train, our y\_train, we pass in our batch size, which we defined earlier as 32, the number of epochs, as well as the

validation set which is going to be our x\_test

and the y\_test, to allow us to evaluate how well we're

actually performing and whether we're over

fitting on that holdout set. You run this, and this

will take just a bit, so I'm going to pause

the video here, and we'll come back when this is done running and

discuss the results. Now our model has run, went through the 10 epochs. As we're starting to learn or probably have

learned at this point, oftentimes it will take a bit for our deep learning models to

actually learn each one of the weights and to optimize on the models

that we're trying to run. Now we're going to call

model\_rnn.evaluate, and we're going to

evaluate on our test set, so on our x\_test and

y\_test we call evaluate, this will take just a second, it's not too long, and we're going to get our

score and our accuracy. That score is just going to be our binary cross entropy loss, so our loss score. Then our accuracy will just

be our actual accuracy. We have all these lines here, we're going to scroll

down to the bottom since we've printed it out, and we see that we have a score, that log loss of 0.45, and then a test accuracy of 0.78. That closes out this video. In the next video we're

going to briefly touch on different ways that

we can manipulate the models that we

just went through, trying different parameters

and hyper parameters, we're not going to go through all possible

different parameters, hyper parameters, but

we will discuss them, and after we go through

it in the next video, I suggest that you as well at home go through

and try playing around with each one of

the different parameters. I'll see you there.

Welcome back. Now, in this exercise we're just going to play around with some

of the parameters, show you some of the parameters that you can play around with. Then on your own, as

mentioned earlier, I would suggest that you

try playing around as well with these max features

and max length, as well as something

that we won't do the hidden dimension for RNN. I believe we're also

going to work with the word embedding dimension, and we'll see the performance

for each as we move along. First thing that

we have is we set the max features equal to 20,000, which is same as

what we had before. Then the other thing

that we have is that we're setting the max length. Recall that we cap off our sequences or our

sentences at a certain length, and then pad them

accordingly as well. Here rather than the

30 that we had before, we're going to pad

or truncate at 80. Everything else that we

have here stays the same. The same holds for

our hidden dimension, as well as our word embeddings, as well as the setup

of our RNN model. We're just going

to run this, we're not going to walk

through that again. Again, we're going to use RMSProp and again the loss will

be binary cross entropy. The metric that we all

track is going to be accuracy along with that loss will automatically be tracked. We'll run this. Then again finally, we

will fit on our training set using our

x\_train and y\_train, and then having that

holdout validation data of x\_test and y\_test. You run this and again, this may take some time to run. The only difference

again that we have with this sample versus before is that we're setting the maxlength

where we will truncate our sequences up here at 80. I'm going to pause

here and we'll come back once this training is done. That will actually take

quite a bit of time to run. We're not also going to run the accuracy results

as we did before, but you can actually see those accuracy results here

on the validation set, and we see that it

went up to 0.842 compared to what we had

before, which was 0.7846. That matches that evaluate

output over here. We see we're able to

increase our accuracy. Now what we want to see again, we're going to play around with just one more parameter here. We have, again, this time

instead of 20,000 features, which is the amount

of actual words we're going to use using the

most common words, we bring that down to 5,000, keeping that maxlength up at 80, and then for our word

embedding dimension, recall that that's going to take those integer values that

we're starting off with, and convert them into X

dimensional vectors here, 20 dimensional prior they were

at 50 dimensional vectors. We're shrinking

that down as well. We're changing two

features actually here. You run this to get our

new x\_train and x\_test. We get our new RNN model. Everything else here the same, and then after that, again, we will use the compile with

the same loss function, the same optimizer,

tracking the same metrics, and we're going to

call fit again here. Then after that, this is again a run and this

will take some time as well. All of these will

take a bit of time. It's part of the process when we're doing this deep-learning. But here we see it's running

through that first epoch, and then we'll do that again

for another 10 epochs here. The goal being that

if the accuracy on our holdout set is

continuing to improve, we should probably want

it for more epochs. We did see that that

was the case up here. But we look at the

validation accuracy, we see that it continued

to go up after each epoch. We probably could

have continued to run that and get even

higher accuracy. We're actually going

to do that here. We'll see after 10 epochs how well we were able to perform, and then after that I'm

just going to run this now. It'll run for another

10 epochs and we'll see how much the accuracy

can actually improve. I'm going to pause

it here and we will get back once we're done

having both these runs, which will be quite

a bit of time. Now, looking at our results

here going through now, 20 epochs, 10 on the first run, another 10 on the next run, and that second 10, of course, as it was in our last notebook, will pick up where we left off. Here we see that

we had actually a 0.8479 on the training set, and we see that

continues to go up as that loss

continues to go down. We see towards the

end that we get that validation accuracy

of about 0.84 that is around equal to what we

were able to accomplish in just 10 epochs using the word embeddings

with 50 dimensions, max features of 20,000, and the max length of 80. I'd say, again, feel free to play around with these

seven parameters, see if you can improve the model. But we are also going

to in the next lecture, start to discuss a more powerful or current

neural net structure with more long-term memory,

specifically LSTMs. I'll see you back in lecture

where we will pick up with long short term memory

models. I'll see you there.