In this section, we're

going to introduce the more complex version of

recurrent neural networks, such as long-short term memory, which we have listed

here or LSTM, as well as gated

recurrent units or GRUs, which are going to be

a bit simpler version but the same concept of LSTMs, as well as other subjects in regards to recurrent

neural networks, which we'll see as we introduce the learning

objectives right now. Our learning goals

for this section; we're going to cover LSTMs and

how they can help to solve that long-term memory

problem that we discussed with

recurrent neural nets. We'll discuss gated

recurrent units or GRUs, which are another solution to that long-term memory problem. That's not quite as

sophisticated as LSTMs, that's going to be more

efficient to train and often have

similar performance. We'll discuss sequence to

sequence models or trying to predict another sequence

given a certain sequence, which will be powerful for language translation and helping us to understand how perhaps words are pieced together

or sequences are pieced together that may be different lengths but

related to one another. Then finally, we're

going to cover some common enterprise

applications of LSTM models, that you may look to use

yourself within the workplace. Now, we discussed how

the matrices that we're using for

recurrent neural nets, tend to weaken the signal of those earlier inputs as we start to get further

down that sequence. What we're ultimately

going to need is some structure that

allows us to keep some portions unchanged over many steps to maintain information from earlier

in the sequence. This is going to be

the problem that our long-short term memory, recurrent neural nets

will go about addressing. Now, the way that LSTM

accomplishes this, is that it makes remembering

easy with a bit more of a complicated update

mechanism for defining our current

neural nets current state. Now by default, we

already know the LSTM should remember information

from that last step, the same as with

recurrent neural nets. But on top of that, rather than keeping or adjusting

pass information, we have more flexibility

in retaining or forgetting a large portion

from those prior steps, besides just that last step. Now, LSTMs are just going to be a special kind of

recurrent neural network. They were invented back in 1997. Is still called state

of the art though, because although the

concept is a bit older, the computing power

that allows it to actually make it applicable

is going to be new. The idea behind it

is that it's going to add it in explicit

memory unit, and the key to that

unit is that it adds on a few additional gate units, which you can think of as gates that allow for

information to be passed along and how long we will continue to allow

that to stay in memory. With that in mind, the cell will have an input gate, which given a certain

value will tell the system to store

that value in memory. Will have " a forget

gate", which again, given a certain value will

determine whether that information will be

removed from memory. Then finally, we're going

to have our output gate, which is going to fire

off the response to move the current hidden unit

forward within our network. Now here we have

our LSTM diagram. If we think back to our

unrolls recurrent neural net, you're probably unrolls,

our original diagram. This is going to be a single cell from that unrolled the LSTM. So we can imagine that we are

working with some sequence, and we're going to have

here just an input of a single word from that

full sequence of words. Now, I know this looks complex, which is what we promise. Well, let's start off

explaining how it works and, I promised by the end of this it'll make it a bit more sense. So we have our input, which is again just going

to be a single word as we were working with

that unroll version, whether that's a single

word or a single time step, we are passing in that input. We also had this c of t, which is our cell state, at time t which will be new and wasn't a part of our

recurrent neural net. We're also going to

have that hidden state, which is something similar to what we saw in the

recurrent neural net, and this will also feed into the next unit along

with that cell state. Then we have our output

which will be the same as the h\_t that we are passing

along to the next cell. And similar to that

recurrent neural net, where at each one of the

cells will have an output, but will ultimately matter, will be that final output from that final cell of

our unrolled version, where we have the

information from all of our inputs

within our sequence. So let's focus now on this new cell state that

we're working with. The cell state's going to

get updated in two stages. We have our forget gate, whose purpose is to give

us an easy mechanism to decide what information

from that prior cell state, as well as the current

input coming in to forget. Then we have the add new

information portion, which tells us what new

information is worth maintaining. So let's start off with

that mechanism built to help us decide what

we should forget. Here, our cell is looking

at values h\_t minus 1, that input of h\_t minus

1 from the prior cell, as well as our new input x\_t. That's again, based off

of the previous output, that h\_ t minus 1 as well as the current input, which is x\_t. We can concatenate these

two vectors together, the x\_t and h\_t minus

1 as we see here, so we have a single vector. Once we have that single vector, looking back at our

equation of x\_t, we see that we take that single vector,

multiply it through, do some transformation using W\_f, some transformation, and pass that through

a sigmoid function. It will output some value

between zero and one. Now let's walk through

the portion built for adding in new information. That's this add in

new information. Here, we are again calculating that same function of sigmoid of a weight matrix of

the concatenation of our input x\_t as

well as h\_t minus 1. Here, of course just

learning new weights. Now we're working with

W\_i rather than W\_f. But again, outputting

some value between zero and one as we pass

that through the sigmoid. With that, we're also going to be computing at this point, a tanh of another weight matrix

and the concatenations of the weight matrix multiplied

by a concatenation of again that input x\_t

and h\_t minus 1. Then we'll pass these values

through a tanh activation. We'll have resulting values

between negative one and one. Then if you look just above our sigmoid and tanh functions

that we just walk through, we have a multiplication

which is meant to multiply those

two values together. The idea being that the tanh is the actual information you are deciding whether

or not to add on. Then that sigmoid between

zero and one will tell you ideally what portion of that new information we

would want to add on. If it's close to one, we

add on all information. If it's close to zero, we don't add on very much

of that new information. Now, if we've been looking

at this diagram throughout, we would notice that the arrows from both our forget portion as well as our add on portion

that we just discussed, will both lead up to our cell state that

we're trying to compute. Our new cell state then will be a function of each

of these outputs. The f\_i that we calculated, which is again that value

between zero and one, will help tell us how

much of that old cell, if we're multiplying it by some value between zero and one, how much to keep and

how much do forget. Then from there we can

add on the output of our last calculation of that tanh and sigmoid

multiplied together, decide how much new

information to add on. That's going to be that

addition function to ultimately get to our

new cell state c\_t, which again is just

going to be f\_i, some value between zero and one, multiplied by our prior cell

state and then adding on i\_t multiplied by that

c\_t that we just calculated to figure out how much extra

information to add on. Now, let's close out by

looking at that output. We have that similar

function again, of taking the sigmoid

of some weight matrix, and this time that

weight matrix is W\_o, so not the same weight matrix, multiplied by the concatenation

of h\_t minus 1 and x\_t. Then that value

will be multiplied by the tanh of our new

updated cell state, that cell state that

we just calculated. Again, we just computed this c\_t, so there's no new weights

that are needed here. We just need to pass that

through the tanh activation. Then multiplying

those two together, we get our h\_t, our hidden state at

the current cell. That's going to be

both the output value which we see here at the top, as well as the input or that h\_t minus 1 as we move

along to the next cell. We see that process here, how that cell state

and the h\_t persist as we input each value

into the sequence. As we've seen, the

well LSTM is going to require a good amount of parameters and thus

a lot of memory, in order to compute this

LSTM that allows us to have this longer term memory deciding what to keep

and what to forget. In the next video, we're going to start

off by discussing GRUs, which may not always

be quite as accurate, but usually does result in

similar results while using much less memory than

needed for the amount of parameters that are

needed here in LSTM. All right, I'll see you there.

Now let's discuss that gated

recurrent unit, or the GRU. Now just to know, we're not going to be

getting as much into the nitty-gritty of the GRU compared to what

we did with the LSTM. But we will highlight that it has similar

functionality overall to the LSTM. Now what makes the difference? Some major differences include

that lack of a cell state. So we saw that cell state

throughout our LSTM. Here, as we see in the diagram, it's just

going to have that past hidden state. And that hidden state will allow for

persistence as well as understanding what will be updated and

what will be forgotten. And as mentioned in the prior video,

ultimately, we can think of the GRU as

a simpler version of the LSTM. As it's still going to accomplish that

same functionality of having a longer term memory than our vanilla

recurrent neural net. It just will have less weight that will

have to keep in memory throughout. So in our GRU,

we're going to have our reset gate again. Which if we look at the diagram

where we see that rt, within that box we just highlighted, that it takes in past hidden state,

ht-1, as well as xt. And passes that through a sigmoid in order

to figure out what's going to be reset. We then have our update gate,

which if we look at the zt, we can see that again,

this is a combination of the ht-1. And our input xt pass through a sigmoid. And there's going to be a lot

more to that interstate. But again, the idea remains the same

of having some type of functionality to decide what will you remember and

maintaining information from the past. And another portion of the cell for updating the cell with

that new information. So the question arises,

shall we use an LSTM or GRU? LSTMs are going to be

a bit more complex and may therefore be able to find

more complicated patterns. And of course, on the other side,

GRUs are going to be a bit simpler and therefore quicker to train. In general, GRUs will perform just about

as well as LSTMs with that shorter training time, especially when

working with smaller datasets. And luckily in Keras, if you're trying to

decide whether to use an LSTM or a GRU, you all we need to do is

call that layer type. And it wouldn't be too complicated to

write up changes between the two and plug and play between the two. Now I want to discuss another concept,

so we're moving away from the LSTM or GRU, but it will be built off of

the idea of recurrent neural nets. I want to discuss this concept of

Seq2Seq or sequence to sequence, which is meant to convert

a sequence from one domain, say here English,

to some other domain such as French. And thus, given the examples I just gave,

obviously, this is going to be very powerful for

machine translation. And if we think about how our recurrent

neural nets work, recall that given a sequence as the words are entered

into the network one at a time. And we see these words coming in. We will have a new updated hidden

state that will have accounted for all the past information. And that's what we see above with h1, h2, through h6 as the sentence The black cat

drank milk was fed through our network. And at the end of our sentence, that final hidden state should have

all the information relating to all the words contained within

that sequence within our sentence. And we can leverage this vector,

that hidden state. As no matter the size of our sentence, if we're just looking at that h6,

that final hidden state, which is just going to be a vector

the size of that state vector. We can take that final hidden state

that contains all the information for that given sequence,

here the English sentence. And that information from the English

words will be what we call the encoder portion of

a encoder-decoder model which is going to be the crux of how we

work with the Seq2Seq modeling. So we have again these words coming in and

this is towards the end of the sentence, drank milk, and then we actually

have a term for end of sentence. And from here, we have our hidden

state from the encoder portion. And now for the decoder portion,

it can now work as a language model, just moving forward. That's just trying to

predict the next word. And it's going to use as its initial state

what was output from the encoder portion. And this makes sure that it's not just a

language model spitting out French words. But it's going to be spitting out new

French words in a sequence conditional on that English sentence.

Now we went over those basics of the encoder decoder model for a sequence to

sequence learning. But maybe you

noticed that the way that this is currently

constructed, the model is going to be producing a single

word at a time, and that single word that's

being produced will be conditional on whatever

that prior word that was produced was. With that in mind, if at any point it

produces one wrong word, we may end up with a completely

different trajectory, and that would throw off

the entire translation and that entire sequence that

we're trying to predict. Now again, the way it's currently constructed

is that it will continue to predict new words until it hits this

end of sentence term. Now, a new solution to solve

for the problem that we just discussed that our

trajectory can be thrown off, would be to produce multiple different

sentences through to the end of sentence

term that we have, and then see which one of those full sentences or full sequences is

the most likely. We can imagine that each of these that we're

seeing being built out here would lead to

some end of sentence term, and given each one of

these different bunches, and once we have some predetermined amount

of possible sentences, possible sequences, we can

then probabilistically determine which one of these full sentences

is the most likely. The way that our

encoder decoder model is currently built out, our decoder works with that hidden state from the

encoder that has information about the entire

sentence.The final hidden state is used as that

initializer for that decoder, whether we're using

that Beam Search that we just discussed, where we can look at multiple

different sentences and decide the most likely or if

we're just looking at one. With that in mind, each

decoder time-step. As we're going through producing each one of

the different words, each one will be depending on that same encoder

embedding and will have no relationship as to

where in the sentence we currently are within the

decoder in regards to, are we at the translation for the cats or we in

translation for drank milk? We'd want some way of

rather than looking at the entire sentence

within the encoder, only looking at

those terms that are similar to the terms where

we're at within the decoder. The attention is going to solve for that problem and allow us to look specifically at

those terms that matter. Now with attention, our

goal is to consider the words that are

most similar to our current position in

our sentence generation. So rather than just using the hidden state from

that entire sentence. Again, just using H6, that final hidden state, we can actually use the hidden state from each

one of the different terms. How does this work? We

know that each word in either language will be represented by some

type of vector. What we can do is that each

word within our decoder, we can look to see how close that vector in our decoder in that different language is to each word within our encoder. We'll have some function s. We have that function s of i j, which we can just think of

as some type of function s, which gives the similarity

measure between the decoder state i and the encoder state j so that we know how

similar each term. You see this is a

function mapping to all the encoder terms to each one of our

single decoder terms to decide which one

is the most similar, and then this similarity

function will then weight the different

embedding layers, each one of these hidden

states from the encoder to give us a better embedding for the prediction of that next word. If the second term in

the encoder is the closest in regards to the

vector distance that S measure, to our decoder term

where we currently are, then that will have a

much higher weight, all the weights adding

to one then terms 1, 3, 4 and 5. This will then

better allow you to translate between different

languages when that ordering of the words are often

different so that you know even though we started

with the cat and friendship, terms would have the cat at

the end of the sentence. Would add that noun at

the end of the sentence, we can still see how close

each one of those terms or each one of those final terms are compared to

our encoder model. That closes out our discussion

of seek to seek models, and I want to talk about here some common business applications for LSTMs or GRUs as well. Those will be interchangeable in regards to what I'm

discussing here. We can use it for forecasting, where LSTM is among most common deep-learning

models used in forecasting in general,

if you are trying to. Predicts how much inventory

you're going to need, how many sales you're

going to have, whether or not the stock

will go up and down. LSTM will be very powerful for forecasting and is the most

used deep learning model. You can use it for

speech recognition in terms of understanding

sequences of words that may be coming

out of my mouth right now. As we just saw, it can be

used for machine translation. We can use it for

image captioning and producing words to describe

an image coming in. We can use it for

question answering. If you're having a call service or customer service

calls coming in, perhaps you can use some

type of language processing such as LSTMs in order

to produce answers. We can use it for

anomaly detection to see if it works with

the current sequence. In regards to maybe there's

some type of seasonality or some types of spike that you should expect or

shouldn't expect, and we should be able to

detect that using LSTMs, as well as robotic

control over trying to move in any type of

step-by-step sequence. That closes out our video here, and just to recap. In this section, we introduced Long Short-Term Memory

Networks and how we can use that cell state in order to

decide whether we're going to forget certain information or keep certain information. We then moved on to Gated

Recurrent Units or GRUs, and how that's somewhat of a simplified version of the

LSTMs that should be able to perform more efficiently

but maybe won't be quite as exact as LSTMs. We then discussed sequence to sequence models or seek to seek, which allow you to go from one type of sequence

to another type of sequence when you're trying

to predict between sequences. Then finally, we

just closed out with some common enterprise

applications of where you can use LSTM

models in practice. That closes out our

section here on LSTMs, as well as our discussion on sequence modeling in general. In the next video, we're going to introduce our first unsupervised

deep-learning model called auto-encoders.

I'll see you there.

# Summary/Review

## **Recurrent Neural Networks (RNNs)**

Recurrent Neural Networks are a class of neural networks that allow previous outputs to be used as inputs while having hidden states. They are mostly used in applications of natural language processing and speech recognition.

One of the main motivations for RNNs is to derive insights from text and do better than “bag of words” implementations. Ideally, each word is processed or understood in the appropriate context.

Words should be handled differently depending on “context”. Also, each word should update the context.

Under the notion of recurrence, words are input one by one. This way, we can handle variable lengths of text. This means that the response to a word depends on the words that preceded it.

These are the two main outputs of an RNN:

* Prediction: What would be the prediction if the sequence ended with that word.
* State: Summary of everything that happened in the past.

**Mathematical Details**

Mathematically, there are cores and subsequent dense layers

current state = function1(old state, current input).

current output = function2(current state).

We learn function1 and function2 by training our network!

r = dimension of input vector

s = dimension of hidden state

t = dimension of output vector (after dense layer)

U is a s × r matrix

W is a s × s matrix

V is a t × s matrix

In which the weight matrices U, V, W are the same across all positions

**Practical Details**

Often, we train on just the ”final” output and ignore intermediate outputs.

Slight variation called Backpropagation Through Time (BPTT) is used to train RNNs.

Sensitive to length of sequence (due to “vanishing/exploding gradient” problem).

In practice, we still set a maximum length to our sequences. If the input is shorter than maximum, we “pad” it. If the input is longer than maximum, we truncate it.

**RNN Applications**

RNNs often focus on text applications, but are commonly used for other sequential data:

* Forecasting: Customer Sales, Loss Rates, Network Traffic.
* Speech Recognition: Call Center Automation, Voice Applications.
* Manufacturing Sensor Data
* Genome Sequences

**Long-Short Term Memory RNNs (LSTM)**

LSTMs are a special kind of RNN (invented in 1997). LSTM has as motivation solve one of the main weaknesses of RNNs, which is that its transitional nature, makes it hard to keep information from distant past in current memory without reinforcement.

LSTM have a more complex mechanism for updating the state.

Standard RNNs have poor memory because the transition Matrix necessarily weakens signal.

This is the problem addressed by Long-Short Term Memory RNNs (LSTM).

To solve it, you need a structure that can leave some dimensions unchanged over many steps.

* By default, LSTMs remember the information from the last step.
* Items are overwritten as an active choice.

The idea for updating states that RNNs use is old, but the available computing power to do it sequence to sequence mapping, explicit memory unit, and text generation tasks is relatively new.

Augment RNNs with a few additional Gate Units:

* Gate Units control how long/if events will stay in memory.
* Input Gate: If its value is such, it causes items to be stored in memory.
* Forget Gate: If its value is such, it causes items to be removed from memory.
* Output Gate: If its value is such, it causes the hidden unit to feed forward (output) in the network.

**Gated Recurrent Units (GRUs)**

GRUs are a gating mechanism for RNNs that is an alternative to LSTM. It is based on the principle of Removed Cell State:

* Past information is now used to transfer past information.
* Think of as a “simpler” and faster version of LSTM.

These are the gates of GRU:

Reset gate: helps decide how much past information to forget.

Update gate: helps decide what information to throw away and what new information to keep.

**LSTM vs GRU**

LSTMs are a bit more complex and may therefore be able to find more complicated patterns.

Conversely, GRUs are a bit simpler and therefore are quicker to train.

GRUs will generally perform about as well as LSTMs with shorter training time, especially for smaller datasets.

In Keras it is easy to switch from one to the other by specifying a layer type. It is relatively quickly to change one for the other.

**Sequence-to-Sequence Models (Seq2Seq)**

Thinking back to any type of RNN interprets text, the model will have a new hidden state at each step of the sequence containing information about all past words.

Seq2Seq improve keeping necessary information in the hidden state from one sequence to the next.

This way, at the end of a sentence, the hidden state will have all information relating to past words.

The size of the vector from the hidden state is the same no matter the size of the sentence.

In a nutshell, there is an encoder, a hidden state, and a decoder.

**Beam Search**

Beam search is an attempt to solve greedy inference.

* Greedy Inference, which means that a model producing one word at a time implies that if it produces one wrong word, it might output a wrong entire sequence of words.
* Beam search tries to produce multiple different hypotheses to produce words until <EOS> and then see which full sentence is most likely.

These are examples of common enterprise applications of LSTM models:

* Forecasting: (LSTM among most common Deep Learning models used in forecasting).
* Speech Recognition
* Machine Translation
* Image Captioning
* Question Answering
* Anomaly Detection
* Robotic Control