Lecture 10 - Recurrent Neural Networks

Okay.

Can everyone hear me?

Okay.

Sorry for the delay.

I had a bit of technical difficulty.

Today was the first time I was trying to use my new touch bar Mac book pro for presenting, and none of the adapters are working.

So, I had to switch laptops at the last minute.

So, thanks.

Sorry about that.

So, today is lecture 10.

We're talking about recurrent neural networks.

So, as of, as usual, a couple administrative notes.

So, we're working hard on assignment one grading.

Those grades will probably be out sometime later today.

Hopefully, they can get out before the A2 deadline.

That's what I'm hoping for.

On a related note, Assignment two is due today at 11:59 p.

m.

so, who's done with that already?

About half you guys.

So, you remember, I did warn you when the assignment went out that it was quite long, to start early.

So, you were warned about that.

But, hopefully, you guys have some late days left.

Also, as another reminder, the midterm will be in class on Tuesday.

If you kind of look around the lecture hall, there are not enough seats in this room to seat all the enrolled students in the class.

So, we'll actually be having the midterm in several other lecture halls across campus.

And we'll be sending out some more details on exactly where to go in the next couple of days.

So a bit of a, another bit of announcement.

We've been working on this sort of fun bit of extra credit thing for you to play with that we're calling the training game.

This is this cool browser-based experience, where you can go in and interactively train neural networks and tweak the hyper parameters during training.

And this should be a really cool interactive way for you to practice some of these hyper parameters tuning skills that we've been talking about the last couple of lectures.

So, this is not required, but this, I think, will be a really useful experience to gain a little bit more intuition into how some of these hyper parameters work for different types of data sets in practice.

So, we're still working on getting all the bugs worked out of this setup, and we'll probably send out some more instructions on exactly how this will work in the next couple of days.

But again, not required.

But please do check it out.

I think it'll be really fun and a really cool thing for you to play with.

And will give you a bit of extra credit if you do some, if you end up working with this and doing a couple of runs with it.

So, we'll again send out some more details about this soon once we get all the bugs worked out.

As a reminder, last time we were talking about CNN Architectures.

We kind of walked through the time line of some of the various winners of the image net classification challenge, kind of the breakthrough result.

As we saw was the AlexNet architecture in 2012, which was a nine layer convolutional network.

It did amazingly well, and it sort of kick started this whole deep learning revolution in computer vision, and kind of brought a lot of these models into the mainstream.

Then we skipped ahead a couple years, and saw that in 2014 image net challenge, we had these two really interesting models, VGG and GoogLeNet, which were much deeper.

So VGG was, they had a 16 and a 19 layer model, and GoogLeNet was, I believe, a 22 layer model.

Although one thing that is kind of interesting about these models is that the 2014 image net challenge was right before batch normalization was invented.

So at this time, before the invention of batch normalization, training these relatively deep models of roughly twenty layers was very challenging.

So, in fact, both of these two models had to resort to a little bit of hackery in order to get their deep models to converge.

So, for VGG, they had the 16 and 19 layer models, but actually they first trained an 11 layer model, because that was what they could get to converge.

And then added some extra random layers in the middle and then continued training, actually training the 16 and 19 layer models.

So, managing this training process was very challenging in 2014 before the invention of batch normalization.

Similarly, for GoogLeNet, we saw that GoogLeNet has these auxiliary classifiers that were stuck into lower layers of the network.

And these were not really needed for the class to, to get good classification performance.

This was just sort of a way to cause extra gradient to be injected directly into the lower layers of the network.

And this sort of, this again was before the invention of batch normalization and now once you have these networks with batch normalization, then you no longer need these slightly ugly hacks in order to get these deeper models to converge.

Then we also saw in the 2015 image net challenge was this really cool model called ResNet, these residual networks that now have these shortcut connections that actually have these little residual blocks where we're going to take our input, pass it through the residual blocks, and then add that output of the, then add our input to the block, to the output from these convolutional layers.

This is kind of a funny architecture, but it actually has two really nice properties.

One is that if we just set all the weights in this residual block to zero, then this block is competing the identity.

So in some way, it's relatively easy for this model to learn not to use the layers that it doesn't need.

In addition, it kind of adds this interpretation to L2 regularization in the context of these neural networks, cause once you put L2 regularization, remember, on your, on the weights of your network, that's going to drive all the parameters towards zero.

And maybe your standard convolutional architecture is driving towards zero.

Maybe it doesn't make sense.

But in the context of a residual network, if you drive all the parameters towards zero, that's kind of encouraging the model to not use layers that it doesn't need, because it will just drive those, the residual blocks towards the identity, whether or not needed for classification.

The other really useful property of these residual networks has to do with the gradient flow in the backward paths.

If you remember what happens at these addition gates in the backward pass, when upstream gradient is coming in through an addition gate, then it will split and fork along these two different paths.

So then, when upstream gradient comes in, it'll take one path through these convolutional blocks, but it will also have a direct connection of the gradient through this residual connection.

So then when you look at, when you imagine stacking many of these residual blocks on top of each other, and our network ends up with hundreds of, potentially hundreds of layers.

Then, these residual connections give a sort of gradient super highway for gradients to flow backward through the entire network.

And this allows it to train much easier and much faster.

And actually, allows these things to converge reasonably well, even when the model is potentially hundreds of layers deep.

And this idea of managing gradient flow in your models is actually super important everywhere in machine learning.

And super prevalent in recurrent networks as well.

So we'll definitely revisit this idea of gradient flow later in today's lecture.

So then, we kind of also saw a couple other more exotic, more recent CNN architectures last time, including DenseNet and FractalNet, and once you think about these architectures in terms of gradient flow, they make a little bit more sense.

These things like DenseNet and FractalNet are adding these additional shortcut or identity connections inside the model.

And if you think about what happens in the backwards pass for these models, these additional funny topologies are basically providing direct paths for gradients to flow from the loss at the end of the network more easily into all the different layers of the network.

So, I think that, again, this idea of managing gradient flow properly in your CNN Architectures is something that we've really seen a lot more in the last couple of years.

And will probably see more moving forward as more exotic architectures are invented.

We also saw this kind of nice plot, plotting performance of the number of flops versus the number of parameters versus the run time of these various models.

And there's some interesting characteristics that you can dive in and see from this plot.

One idea is that VGG and AlexNet have a huge number of parameters, and these parameters actually come almost entirely from the fully connected layers of the models.

So AlexNet has something like roughly 62 million parameters, and if you look at that last fully connected layer, the final fully connected layer in AlexNet is going from an activation volume of six by six by 256 into this fully connected vector of 496.

So if you imagine what the weight matrix needs to look like at that layer, the weight matrix is gigantic.

It's number of entries is six by six, six times six times 256 times 496.

And if you multiply that out, you see that that single layer has 38 million parameters.

So more than half of the parameters of the entire AlexNet model are just sitting in that last fully connected layer.

And if you add up all the parameters in just the fully connected layers of AlexNet, including these other fully connected layers, you see something like 59 of the 62 million parameters in AlexNet are sitting in these fully connected layers.

So then when we move other architectures, like GoogLeNet and ResNet, they do away with a lot of these large fully connected layers in favor of global average pooling at the end of the network.

And this allows these networks to really cut, these nicer architectures, to really cut down the parameter count in these architectures.

So that was kind of our brief recap of the CNN architectures that we saw last lecture, and then today, we're going to move to one of my favorite topics to talk about, which is recurrent neural networks.

So, so far in this class, we've seen, what I like to think of as kind of a vanilla feed forward network, all of our network architectures have this flavor, where we receive some input and that input is a fixed size object, like an image or vector.

That input is fed through some set of hidden layers and produces a single output, like a classification, like a set of classifications scores over a set of categories.

But in some context in machine learning, we want to have more flexibility in the types of data that our models can process.

So once we move to this idea of recurrent neural networks, we have a lot more opportunities to play around with the types of input and output data that our networks can handle.

So once we have recurrent neural networks, we can do what we call these one to many models.

Or where maybe our input is some object of fixed size, like an image, but now our output is a sequence of variable length, such as a caption.

Where different captions might have different numbers of words, so our output needs to be variable in length.

We also might have many to one models, where our input could be variably sized.

This might be something like a piece of text, and we want to say what is the sentiment of that text, whether it's positive or negative in sentiment.

Or in a computer vision context, you might imagine taking as input a video, and that video might have a variable number of frames.

And now we want to read this entire video of potentially variable length.

And then at the end, make a classification decision about maybe what kind of activity or action is going on in that video.

We also have a, we might also have problems where we want both the inputs and the output to be variable in length.

We might see something like this in machine translation, where our input is some, maybe, sentence in English, which could have a variable length, and our output is maybe some sentence in French, which also could have a variable length.

And crucially, the length of the English sentence might be different from the length of the French sentence.

So, we need some models that have the capacity to accept both variable length sequences on the input and on the output.

Finally, we might also consider problems where our input is variably length, like something like a video sequence with a variable number of frames.

And now we want to make a decision for each element of that input sequence.

So in the context of videos, that might be making some classification decision along every frame of the video.

And recurrent neural networks are this kind of general paradigm for handling variable sized sequence data that allow us to pretty naturally capture all of these different types of setups in our models.

So recurring neural networks are actually important, even for some problems that have a fixed size input and a fixed size output.

Recurrent neural networks can still be pretty useful.

So in this example, we might want to do, for example, sequential processing of our input.

So here, we're receiving a fixed size input like an image, and we want to make a classification decision about, like, what number is being shown in this image?

But now, rather than just doing a single feed forward pass and making the decision all at once, this network is actually looking around the image and taking various glimpses of different parts of the image.

And then after making some series of glimpses, then it makes its final decision as to what kind of number is present.

So here, we had one, so here, even though our input and outputs, our input was an image, and our output was a classification decision, even this context, this idea of being able to handle variably length processing with recurrent neural networks can lead to some really interesting types of models.

There's a really cool paper that I like that applied this same type of idea to generating new images.

Where now, we want the model to synthesize brand new images that look kind of like the images it saw in training, and we can use a recurrent neural network architecture to actually paint these output images sort of one piece at a time in the output.

You can see that, even though our output is this fixed size image, we can have these models that are working overtime to compute parts of the output one at a time sequentially.

And we can use recurrent neural networks for that type of setup as well.

So, after this sort of cool pitch about all these cool things that RNNs can do, you might wonder, like what exactly are these things?

So, in general, a recurrent neural network is this little, has this little recurrent core cell and it will take some input x, feed that input into the RNN, and that RNN has some internal hidden state, and that internal hidden state will be updated every time that the RNN reads a new input.

And that internal hidden state will be then fed back to the model the next time it reads an input.

And frequently, we will want our RNN"s to also produce some output at every time step, so we'll have this pattern where it will read an input, update its hidden state, and then produce an output.

So, then the question is what is the functional form of this recurrence relation that we're computing?

So, inside this little green RNN block, we're computing some recurrence relation, with a function f.

So, this function f will depend on some weights, w.

It will accept the previous hidden state, h t - 1, as well as the input at the current state, x t, and this will output the next hidden state, or the updated hidden state, that we call h t.

And now, then as we read the next input, this hidden state, this new hidden state, h t, will then just be passed into the same function as we read the next input, x t plus one.

And now, if we wanted to produce some output at every time step of this network, we might attach some additional fully connected layers that read in this h t at every time step.

And make that decision based on the hidden state at every time step.

And one thing to note is that we use the same function, f w, and the same weights, w, at every time step of the computation.

So, then kind of the simplest function form that you can imagine is what we call this vanilla recurrent neural network.

So here, we have this same functional form from the previous slide, where we're taking in our previous hidden state and our current input and we need to produce the next hidden state.

And the kind of simplest thing you might imagine is that we have some weight matrix, w x h, that we multiply against the input, x t, as well as another weight matrix, w h h, that we multiply against the previous hidden state.

So, we make these two multiplications against our two states, add them together, and squash them through a tanh, so we get some kind of non-linearity in the system.

You might be wondering why we use a tanh here and not some other type of non-linearity?

After all that we've said negative about tanh's in previous lectures, and I think we'll return a little bit to that later on when we talk about more advanced architectures, like lstm.

So then, this, so then, in addition in this architecture, if we wanted to produce some y t at every time step, you might have another weight matrix, w, you might have another weight matrix that accepts this hidden state and then transforms it to some y to produce maybe some class score predictions at every time step.

And when I think about recurrent neural networks, I kind of think about, you can also, you can kind of think of recurrent neural networks in two ways.

One is this concept of having a hidden state that feeds back at itself, recurrently.

But I find that picture a little bit confusing.

And sometimes, I find it clearer to think about unrolling this computational graph for multiple time steps.

And this makes the data flow of the hidden states and the inputs and the outputs and the weights maybe a little bit clearer.

So then at the first-time step, we'll have some initial hidden state h zero.

This is usually initialized to zeros for most context, in most contexts, a then we'll have some input, x t.

This initial hidden state, h zero, and our current input, x t, will go into our f w function.

This will produce our next hidden state, h one.

And then, we'll repeat this process when we receive the next input.

So now our current h one and our x one, will go into that same f w, to produce our next output, h two.

And this process will repeat over and over again, as we consume all of the input, x ts, in our sequence of inputs.

And now, one thing to note, is that we can actually make this even more explicit and write the w matrix in our computational graph.

And here you can see that we're re-using the same w matrix at every time step of the computation.

So now every time that we have this little f w block, it's receiving a unique h and a unique x, but all of these blocks are taking the same w.

And if you remember, we talked about how gradient flows in back propagation, when you re-use the same, when you re-use the same node multiple times in a computational graph, then remember during the backward pass, you end up summing the gradients into the w matrix when you're computing a d los d w.

So, if you kind of think about the back propagation for this model, then you'll have a separate gradient for w flowing from each of those time steps, and then the final gradient for w will be the sum of all of those individuals per time step gradients.

We can also write to this y t explicitly in this computational graph.

So then, this output, h t, at every time step might feed into some other little neural network that can produce a y t, which might be some class scores, or something like that, at every time step.

We can also make the loss more explicit.

So in many cases, you might imagine producing, you might imagine that you have some ground truth label at every time step of your sequence, and then you'll compute some loss, some individual loss, at every time step of these outputs, y t's.

And this loss might, it will frequently be something like soft max loss, in the case where you have, maybe, a ground truth label at every time step of the sequence.

And now the final loss for the entire, for this entire training stop, will be the sum of these individual losses.

So now, we had a scaler loss at every time step?

And we just summed them up to get our final scaler loss at the top of the network.

And now, if you think about, again, back propagation through this thing, we need, in order to train the model, we need to compute the gradient of the loss with respect to w.

So, we'll have loss flowing from that final loss into each of these time steps.

And then each of those time steps will compute a local gradient on the weights, w, which will all then be summed to give us our final gradient for the weights, w.

Now if we have a, sort of, this many to one situation, where maybe we want to do something like sentiment analysis, then we would typically make that decision based on the final hidden state of this network.

Because this final hidden state kind of summarizes all of the context from the entire sequence.

Also, if we have a kind of a one to many situation, where we want to receive a fix sized input and then produce a variably sized output.

Then you'll commonly use that fixed size input to initialize, somehow, the initial hidden state of the model, and now the recurrent network will tick for each cell in the output.

And now, as you produce your variably sized output, you'll unroll the graph for each element in the output.

So this, when we talk about the sequence to sequence models where you might do something like machine translation, where you take a variably sized input and a variably sized output.

You can think of this as a combination of the many to one, plus a one to many.

So, we'll kind of proceed in two stages, what we call an encoder and a decoder.

So if you're the encoder, we'll receive the variably sized input, which might be your sentence in English, and then summarize that entire sentence using the final hidden state of the encoder network.

And now we're in this many to one situation where we've summarized this entire variably sized input in this single vector, and now, we have a second decoder network, which is a one to many situation, which will input that single vector summarizing the input sentence and now produce this variably sized output, which might be your sentence in another language.

And now in this variably sized output, we might make some predictions at every time step, maybe about what word to use.

And you can imagine kind of training this entire thing by unrolling this computational graph summing the losses at the output sequence and just performing back propagation, as usual.

So as a bit of a concrete example, one thing that we frequently use recurrent neural networks for, is this problem called language modeling.

So, in the language modeling problem, we want to read some sequence of, we want to have our network, sort of, understand how to produce natural language.

So, in the, so this, this might happen at the character level where our model will produce characters one at a time.

This might also happen at the word level where our model will produce words one at a time.

But in a very simple example, you can imagine this character level language model where we want, where the network will read some sequence of characters and then it needs to predict, what will the next character be in this stream of text?

So, in this example, we have this very small vocabulary of four letters, h, e, l, and o, and we have this example training sequence of the word hello, h, e, l, l, o.

So, during training, when we're training this language model, we will feed the characters of this training sequence as inputs, as x ts, to out input of our, we'll feed the characters of our training sequence, these will be the x ts that we feed in as the inputs to our recurrent neural network.

And then, each of these inputs, it's a letter, and we need to figure out a way to represent letters in our network.

So, what we'll typically do is figure out what is our total vocabulary.

In this case, our vocabulary has four elements.

And each letter will be represented by a vector that has zeros in every slot but one, and a one for the slot in the vocabulary corresponding to that letter.

In this little example, since our vocab has the four letters, h, e, l, o, then our input sequence, the h is represented by a four-element vector with a one in the first slot and zero's in the other three slots.

And we use the same sort of pattern to represent all the different letters in the input sequence.

Now, during this forward pass of what this network is doing, at the first-time step, it will receive the input letter h.

That will go into the first RNN, to the RNN cell, and then we'll produce this output, y t, which is the network making predictions about for each letter in the vocabulary, which letter does it think is most likely going to come next.

In this example, the correct output letter was e because our training sequence was hello, but the model is actually predicting, I think it's actually predicting o as the most likely letter.

So, in this case, this prediction was wrong and we would use softmaxt loss to quantify our unhappiness with these predictions.

The next time step, we would feed in the second letter in the training sequence, e, and this process will repeat.

We'll now represent e as a vector.

Use that input vector together with the previous hidden state to produce a new hidden state and now use the second hidden state to, again, make predictions over every letter in the vocabulary.

In this case, because our training sequence was hello, after the letter e, we want our model to predict l.

In this case, our model may have very low predictions for the letter l, so we would incur high loss.

And you kind of repeat this process over and over, and if you train this model with many different sequences, then eventually it should learn how to predict the next character in a sequence based on the context of all the previous characters that it's seen before.

And now, if you think about what happens at test time, after we train this model, one thing that we might want to do with it is a sample from the model, and actually use this trained neural network model to synthesize new text that kind of looks similar in spirit to the text that it was trained on.

The way that this will work is we'll typically see the model with some input prefix of text.

In this case, the prefix is just the single letter h, and now we'll feed that letter h through the first-time step of our recurrent neural network.

It will product this distribution of scores over all the characters in the vocabulary.

Now, at training time, we'll use these scores to actually sample from it.

So, we'll use a softmaxt function to convert those scores into a probability distribution and then we will sample from that probability distribution to actually synthesize the second letter in the sequence.

And in this case, even though the scores were pretty bad, maybe we got lucky and sampled the letter e from this probability distribution.

And now, we'll take this letter e that was sampled from this distribution and feed it back as input into the network at the next time step.

Now, we'll take this e, pull it down from the top, feed it back into the network as one of these, sort of, one hot vectoral representations, and then repeat the process in order to synthesize the second letter in the output.

And we can repeat this process over and over again to synthesize a new sequence using this trained model, where we're synthesizing the sequence one character at a time using these predicted probability distributions at each time step.

Question?

Yeah, that's a great question.

So the question is why might we sample instead of just taking the character with the largest score?

In this case, because of the probability distribution that we had, it was impossible to get the right character, so we had the sample so the example could work out, and it would make sense.

But in practice, sometimes you'll see both.

So sometimes you'll just take the argmax probability, and that will sometimes be a little bit more stable, but one advantage of sampling, in general, is that it lets you get diversity from your models.

Sometimes you might have the same input, maybe the same prefix, or in the case of image captioning, maybe the same image.

But then if you sample rather than taking the argmax, then you'll see that sometimes these trained models are actually able to produce multiple different types of reasonable output sequences, depending on the kind, depending on which samples they take at the first time steps.

It's actually kind of a benefit cause we can get now more diversity in our outputs.

Another question?

Could we feed in the softmax vector instead of the one element vector?

You mean at test time?

Yeah, so the question is, at test time, could we feed in this whole softmax vector rather than a one hot vector?

There's kind of two problems with that.

One is that that's very different from the data that it saw at training time.

In general, if you ask your model to do something at test time, which is different from training time, then it'll usually blow up.

It'll usually give you garbage and you'll usually be sad.

The other problem is that in practice, our vocabularies might be very large.

So maybe, in this simple example, our vocabulary is only four elements, so it's not a big problem.

But if you're thinking about generating words one at a time, now your vocabulary is every word in the English language, which could be something like tens of thousands of elements.

So in practice, this first element, this first operation that's taking in this one hot vector, is often performed using sparse vector operations rather than dense factors.

It would be, sort of, computationally really bad if you wanted to have this load of 10,000 elements softmax vector.

So that's usually why we use a one hot instead, even at test time.

This idea that we have a sequence and we produce an output at every time step of the sequence and then finally compute some loss, this is sometimes called backpropagation through time because you're imagining that in the forward pass, you're kind of stepping forward through time and then during the backward pass, you're sort of going backwards through time to compute all your gradients.

This can actually be kind of problematic if you want to train the sequences that are very, very long.

So if you imagine that we were kind of trying to train a neural network language model on maybe the entire text of Wikipedia, which is, by the way, something that people do pretty frequently, this would be super slow, and every time we made a gradient step, we would have to make a forward pass through the entire text of all of wikipedia, and then make a backward pass through all of wikipedia, and then make a single gradient update.

And that would be super slow.

Your model would never converge.

It would also take a ridiculous amount of memory so this would be just really bad.

In practice, what people do is this, sort of, approximation called truncated backpropagation through time.

Here, the idea is that, even though our input sequence is very, very long, and even potentially infinite, what we'll do is that during, when we're training the model, we'll step forward for some number of steps, maybe like a hundred is kind of a ballpark number that people frequently use, and we'll step forward for maybe a hundred steps, compute a loss only over this sub sequence of the data, and then back propagate through this sub sequence, and now make a gradient step.

And now, when we repeat, well, we still have these hidden states that we computed from the first batch, and now, when we compute this next batch of data, we will carry those hidden states forward in time, so the forward pass will be exactly the same.

But now when we compute a gradient step for this next batch of data, we will only backpropagate again through this second batch.

Now, we'll make a gradient step based on this truncated backpropagation through time.

This process will continue, where now when we make the next batch, we'll again copy these hidden states forward, but then step forward and then step backward, but only for some small number of time steps.

So, this is, you can kind of think of this as being an alegist who's the cast at gradient descent in the case of sequences.

Remember, when we talked about training our models on large data sets, then these data sets, it would be super expensive to compute the gradients over every element in the data set.

So instead, we kind of take small samples, small mini batches instead, and use mini batches of data to compute gradient stops in any kind of image classification case.

Question?

Is this kind of, the question is, is this kind of making the Mark Hobb assumption?

No, not really.

Because we're carrying this hidden state forward in time forever.

It's making a Macrobian assumption in the sense that, conditioned on the hidden state, but the hidden state is all that we need to predict the entire future of the sequence.

But that assumption is kind of built into the recurrent neural network formula from the start.

And that's not really particular to back propagation through time.

Back propagation through time, or sorry, truncated back prop though time is just the way to approximate these gradients without going making a backwards pass through your potentially very large sequence of data.

This all sounds very complicated and confusing and it sounds like a lot of code to write, but in fact, this can acutally be pretty concise.

Andrea has this example of what he calls min-char-rnn, that does all of this stuff in just like 112 lines of Python.

It handles building the vocabulary.

It trains the model with truncated back propagation through time.

And then, it can actually sample from that model in actually not too much code.

So even though this sounds like kind of a big, scary process, it's actually not too difficult.

I'd encourage you, if you're confused, to maybe go check this out and step through the code on your own time, and see, kind of, all of these concrete steps happening in code.

So this is all in just a single file, all using numpy with no dependencies.

This was relatively easy to read.

So then, once we have this idea of training a recurrent neural network language model, we can actually have a lot of fun with this.

And we can take in, sort of, any text that we want.

Take in, like, whatever random text you can think of from the internet, train our recurrent neural network language model on this text, and then generate new text.

So in this example, we took this entire text of all of Shakespeare's works, and then used that to train a recurrent neural network language model on all of Shakespeare.

And you can see that the beginning of training, it's kind of producing maybe random gibberish garbage, but throughout the course of training, it ends up producing things that seem relatively reasonable.

And after you've, after this model has been trained pretty well, then it produces text that seems, kind of, Shakespeare-esque to me.

"Why do what that day," replied, whatever, right, you can read this.

Like, it kind of looks kind of like Shakespeare.

And if you actually train this model even more, and let it converge even further, and then sample these even longer sequences, you can see that it learns all kinds of crazy cool stuff that really looks like a Shakespeare play.

It knows that it uses, maybe, these headings to say who's speaking.

Then it produces these bits of text that have crazy dialogue that sounds kind of Shakespeare-esque.

It knows to put line breaks in between these different things.

And this is all, like, really cool, all just sort of learned from the structure of the data.

We can actually get even crazier than this.

This was one of my favorite examples.

I found online, there's this.

Is anyone a mathematician in this room?

Has anyone taken an algebraic topology course by any chance?

Wow, a couple, that's impressive.

So you probably know more algebraic topology than me, but I found this open source algebraic topology textbook online.

It's just a whole bunch of tech files that are like this super dense mathematics.

And LaTac, cause LaTac is sort of this, let's you write equations and diagrams and everything just using plain text.

We can actually train our recurrent neural network language model on the raw LaTac source code of this algebraic topology textbook.

And if we do that, then after we sample from the model, then we get something that seems like, kind of like algebraic topology.

So, it knows to like put equations.

It puts all kinds of crazy stuff.

It's like, to prove study, we see that F sub U is a covering of x prime, blah, blah, blah, blah, blah.

It knows where to put unions.

It knows to put squares at the end of proofs.

It makes lemmas.

It makes references to previous lemmas.

Right, like we hear, like.

It's namely a bi-lemma question.

We see that R is geometrically something.

So it's actually pretty crazy.

It also sometimes tries to make diagrams.

For those of you that have taken algebraic topology, you know that these commutative diagrams are kind of a thing that you work with a lot So it kind of got the general gist of how to make those diagrams, but they actually don't make any sense.

And actually, one of my favorite examples here is that it sometimes omits proofs.

So it'll sometimes say, it'll sometimes say something like theorem, blah, blah, blah, blah, blah, proof omitted.

This thing kind of has gotten the gist of how some of these math textbooks look like.

We can have a lot of fun with this.

So we also tried training one of these models on the entire source code of the Linux kernel.

Cause again, this character level stuff that we can train on, And then, when we sample this, it actually again looks like C source code.

It knows how to write if statements.

It has, like, pretty good code formatting skills.

It knows to indent after these if statements.

It knows to put curly braces.

It actually even makes comments about some things that are usually nonsense.

One problem with this model is that it knows how to declare variables.

But it doesn't always use the variables that it declares.

And sometimes it tries to use variables that haven't been declared.

This wouldn't compile.

I would not recommend sending this as a pull request to Linux.

This thing also figures out how to recite the GNU, this GNU license character by character.

It kind of knows that you need to recite the GNU license and after the license comes some includes, then some other includes, then source code.

This thing has actually learned quite a lot about the general structure of the data.

Where, again, during training, all we asked this model to do was try to predict the next character in the sequence.

We didn't tell it any of this structure, but somehow, just through the course of this training process, it learned a lot about the latent structure in the sequential data.

Yeah, so it knows how to write code.

It does a lot of cool stuff.

I had this paper with Andre a couple years ago where we trained a bunch of these models and then we wanted to try to poke into the brains of these models and figure out like what are they doing and why are they working.

So we saw, in our, these recurring neural networks has this hidden vector which is, maybe, some vector that's updated over every time step.

And then what we wanted to try to figure out is could we find some elements of this vector that have some Symantec interpretable meaning.

So what we did is we trained a neural network language model, one of these character level models on one of these data sets, and then we picked one of the elements in that hidden vector and now we look at what is the value of that hidden vector over the course of a sequence to try to get some sense of maybe what these different hidden states are looking for.

When you do this, a lot of them end up looking kind of like random gibberish garbage.

So here again, what we've done, is we've picked one element of that vector, and now we run the sequence forward through the trained model, and now the color of each character corresponds to the magnitude of that single scaler element of the hidden vector at every time step when it's reading the sequence.

So, you can see that a lot of the vectors in these hidden states are kind of not very interpretable.

It seems like they're kind of doing some of this low-level language modeling to figure out what character should come next.

But some of them end up quite nice.

So here we found this vector that is looking for quotes.

You can see that there's this one hidden element, this one element in the vector, that is off, off, off, off, off blue and then once it hits a quote, it turns on and remains on for the duration of this quote.

And now when we hit the second quotation mark, then that cell turns off.

So somehow, even though this model was only trained to predict the next character in a sequence, it somehow learned that a useful thing, in order to do this, might be to have some cell that's trying to detect quotes.

We also found this other cell that is, looks like it's counting the number of characters since a line break.

So you can see that at the beginning of each line, this element starts off at zero.

Throughout the course of the line, it's gradually redder, so that value increases.

And then after the new line character, it resets to zero.

So you can imagine that maybe this cell is letting the network keep track of when it needs to write to produce these new line characters.

We also found some that, when we trained on the linux source code, we found some examples that are turning on inside the conditions of if statements.

So this maybe allows the network to differentiate whether it's outside an if statement or inside that condition, which might help it model these sequences better.

We also found some that turn on in comments, or some that seem like they're counting the number of indentation levels.

This is all just really cool stuff because it's saying that even though we are only trying to train this model to predict next characters, it somehow ends up learning a lot of useful structure about the input data.

One kind of thing that we often use, so this is not really been computer vision so far, and we need to pull this back to computer vision since this is a vision class.

We've alluded many times to this image captioning model where we want to build models that can input an image and then output a caption in natural language.

There were a bunch of papers a couple years ago that all had relatively similar approaches.

But I'm showing the figure from the paper from our lab in a totally un-biased way.

But, the idea here is that the caption is this variably length sequence that we might, the sequence might have different numbers of words for different captions.

So, this is a totally natural fit for a recurrent neural network language model.

So, then what this model looks like is we have some convolutional network which will input the, which will take as input the image, and we've seen a lot about how convolution networks work at this point, and that convolutional network will produce a summary vector

of the image which will then feed into the first time step of one of these recurrent neural network language models which will then produce words of the caption one at a time.

So the way that this kind of works at test time after the model is trained looks almost exactly the same as these character level language models that we saw a little bit ago.

We'll take our input image, feed it through our convolutional network.

But now instead of taking the softmax scores from an image net model, we'll instead take this 4,096-dimensional vector from the end of the model, and we'll take that vector and use it to summarize the whole content of the image.

Now, remember when we talked about RNN language models, we said that we need to see the language model with that first initial input to tell it to start generating text.

So in this case, we'll give it some special start token, which is just saying, hey, this is the start of a sentence.

Please start generating some text conditioned on this image information.

So now previously, we saw that in this RNN language model, we had these matrices that were taking the previous, the input at the current time step and the hidden state of the previous time step and combining those to get the next hidden state.

Well now, we also need to add in this image information.

So one way, people play around with exactly different ways to incorporate this image information, but one simple way is just to add a third weight matrix that is adding in this image information at every time step to compute the next hidden state.

So now, we'll compute this distribution over all scores in our vocabulary and here, our vocabulary is something like all English words, so it could be pretty large.

We'll sample from that distribution and now pass that word back as input at the next time step.

And that will then feed that word in, again get a distribution over all words in the vocab, and again sample to produce the next word.

So then, after that thing is all done, we'll maybe generate, we'll generate this complete sentence.

We stop generation once we sample the special ends token, which kind of corresponds to the period at the end of the sentence.

Then once the network samples this ends token, we stop generation and we're done and we've gotten our caption for this image.

And now, during training, we trained this thing to generate, like we put an end token at the end of every caption during training so that the network kind of learned during training that end tokens come at the end of sequences.

So then, during test time, it tends to sample these end tokens once it's done generating.

So we trained this model in kind of a completely supervised way.

You can find data sets that have images together with natural language captions.

Microsoft COCO is probably the biggest and most widely used for this task.

But you can just train this model in a purely supervised way.

And then backpropagate through to jointly train both this recurrent neural network language model and then also pass gradients back into this final layer of this the CNN and additionally update the weights of the CNN to jointly tune all parts of the model to perform this task.

Once you train these models, they actually do some pretty reasonable things.

These are some real results from a model, from one of these trained models, and it says things like a cat sitting on a suitcase on the floor, which is pretty impressive.

It knows about cats sitting on a tree branch, which is also pretty cool.

It knows about two people walking on the beach with surfboards.

So these models are actually pretty powerful and can produce relatively complex captions to describe the image.

But that being said, these models are really not perfect.

They're not magical.

Just like any machine learning model, if you try to run them on data that was very different from the training data, they don't work very well.

So for example, this example, it says a woman is holding a cat in her hand.

There's clearly no cat in the image.

But she is wearing a fur coat, and maybe the texture of that coat kind of looked like a cat to the model.

Over here, we see a woman standing on a beach holding a surfboard.

Well, she's definitely not holding a surfboard and she's doing a handstand, which is maybe the interesting part of that image, and the model totally missed that.

Also, over here, we see this example where there's this picture of a spider web in the tree branch, and it totally, and it says something like a bird sitting on a tree branch.

So it totally missed the spider, but during training, it never really saw examples of spiders.

It just knows that birds sit on tree branches during training.

So it kind of makes these reasonable mistakes.

Or here at the bottom, it can't really tell the difference between this guy throwing and catching the ball, but it does know that it's a baseball player and there's balls and things involved.

So again, just want to say that these models are not perfect.

They work pretty well when you ask them to caption images that were similar to the training data, but they definitely have a hard time generalizing far beyond that.

So another thing you'll sometimes see is this slightly more advanced model called Attention, where now when we're generating the words of this caption, we can allow the model to steer it's attention to different parts of the image.

And I don't want to spend too much time on this.

But the general way that this works is that now our convolutional network, rather than producing a single vector summarizing the entire image, now it produces some grid of vectors that summarize the, that give maybe one vector for each spatial location in the image.

And now, when we, when this model runs forward, in addition to sampling the vocabulary at every time step, it also produces a distribution over the locations in the image where it wants to look.

And now this distribution over image locations can be seen as a kind of a tension of where the model should look during training.

So now that first hidden state computes this distribution over image locations, which then goes back to the set of vectors to give a single summary vector that maybe focuses the attention on one part of that image.

And now that summary vector gets fed, as an additional input, at the next time step of the neural network.

And now again, it will produce two outputs.

One is our distribution over vocabulary words.

And the other is a distribution over image locations.

This whole process will continue, and it will sort of do these two different things at every time step.

And after you train the model, then you can see that it kind of will shift it's attention around the image for every word that it generates in the caption.

Here you can see that it produced the caption, a bird is flying over, I can't see that far.

But you can see that its attention is shifting around different parts of the image for each word in the caption that it generates.

There's this notion of hard attention versus soft attention, which I don't really want to get into too much, but with this idea of soft attention, we're kind of taking a weighted

combination of all features from all image locations, whereas in the hard attention case, we're forcing the model to select exactly one location to look at in the image at each time step.

So the hard attention case where we're selecting exactly one image location is a little bit tricky because that is not really a differentiable function, so you need to do something slightly fancier than vanilla backpropagation in order to just train the model in that scenario.

And I think we'll talk about that a little bit later in the lecture on reinforcement learning.

Now, when you look at after you train one of these attention models and then run it on to generate captions, you can see that it tends to focus it's attention on maybe the salient or semantically meaningful part of the image when generating captions.

You can see that the caption was a woman is throwing a frisbee in a park and you can see that this attention mask, when it generated the word, when the model generated the word frisbee, at the same time, it was focusing it's attention on this image region that actually contains the frisbee.

This is actually really cool.

We did not tell the model where it should be looking at every time step.

It sort of figured all that out for itself during the training process.

Because somehow, it figured out that looking at that image region was the right thing to do for this image.

And because everything in this model is differentiable, because we can backpropagate through all these soft attention steps, all of this soft attention stuff just comes out through the training process.

So that's really, really cool.

By the way, this idea of recurrent neural networks and attention actually gets used in other tasks beyond image captioning.

One recent example is this idea of visual question answering.

So here, our model is going to take two things as input.

It's going to take an image and it will also take a natural language question that's asking some question about the image.

Here, we might see this image on the left and we might ask the question, what endangered animal is featured on the truck?

And now the model needs to select from one of these four natural language answers about which of these answers correctly answers that question in the context of the image.

So you can imagine kind of stitching this model together using CNNs and RNNs in kind of a natural way.

Now, we're in this many to one scenario, where now our model needs to take as input this natural language sequence, so we can imagine running a recurrent neural network over each element of that input question, to now summarize the input question in a single vector.

And then we can have a CNN to again summarize the image, and now combine both the vector from the CNN and the vector from the question and coding RNN to then predict a distribution over answers.

We also sometimes, you'll also sometimes see this idea of soft special attention being incorporated into things like visual question answering.

So you can see that here, this model is also having the spatial attention over the image when it's trying to determine answers to the questions.

Just to, yeah, question?

So the question is How are the different inputs combined?

Do you mean like the encoded question vector and the encoded image vector?

Yeah, so the question is how are the encoded image and the encoded question vector combined?

Kind of the simplest thing to do is just to concatenate them and stick them into fully connected layers.

That's probably the most common and that's probably the first thing to try.

Sometimes people do slightly fancier things where they might try to have multiplicative interactions between those two vectors to allow a more powerful function.

But generally, concatenation is kind of a good first thing to try.

Okay, so now we've talked about a bunch of scenarios where RNNs are used for different kinds of problems.

And I think it's super cool because it allows you to start tackling really complicated problems combining images and computer vision with natural language processing.

And you can see that we can kind of stitch together these models like Lego blocks and attack really complicated things, Like image captioning or visual question answering just by stitching together these relatively simple types of neural network modules.

But I'd also like to mention that so far, we've talked about this idea of a single recurrent network layer, where we have sort of one hidden state, and another thing that you'll see pretty commonly is this idea of a multilayer recurrent neural network.

Here, this is a three-layer recurrent neural network, so now our input goes in, goes into, goes in and produces a sequence of hidden states from the first recurrent neural network layer.

And now, after we run kind of one recurrent neural network layer, then we have this whole sequence of hidden states.

And now, we can use the sequence of hidden states as an input sequence to another recurrent neural network layer.

And then you can just imagine, which will then produce another sequence of hidden states from the second RNN layer.

And then you can just imagine stacking these things on top of each other, because we know that we've seen in other contexts that deeper models tend to perform better for various problems.

And the same kind of holds in RNNs as well.

For many problems, you'll see maybe a two or three layer recurrent neural network model is pretty commonly used.

You typically don't see super deep models in RNNs.

So generally, like two, three, four layer RNNs is maybe as deep as you'll typically go.

Then, I think it's also really interesting and important to think about, now we've seen kind of what kinds of problems these RNNs can be used for, but then you need to think a little bit more carefully about exactly what happens to these models when we try to train them.

So here, I've drawn this little vanilla RNN cell that we've talked about so far.

So here, we're taking our current input, x t, and our previous hidden state, h t minus one, and then we stack, those are two vectors.

So we can just stack them together.

And then perform this matrix multiplication with our weight matrix, to give our, and then squash that output through a tanh, and that will give us our next hidden state.

And that's kind of the basic functional form of this vanilla recurrent neural network.

But then, we need to think about what happens in this architecture during the backward pass when we try to compute gradients?

So then if we think about trying to compute, so then during the backwards pass, we'll receive the derivative of our h t, we'll receive derivative of loss with respect to h t.

And during the backward pass through the cell, we'll need to compute derivative of loss to the respect of h t minus one.

Then, when we compute this backward pass, we see that the gradient flows backward through this red path.

So first, that gradient will flow backwards through this tanh gate, and then it will flow backwards through this matrix multiplication gate.

And then, as we've seen in the homework and when implementing these matrix multiplication layers, when you backpropagate through this matrix multiplication gate, you end up multiplying by the transpose of that weight matrix.

So that means that every time we backpropagate through one of these vanilla RNN cells, we end up multiplying by some part of the weight matrix.

So now if you imagine that we are sticking many of these recurrent neural network cells in sequence, because again this is an RNN.

We want a model sequence.

Now if you imagine what happens to the gradient flow through a sequence of these layers, then something kind of fishy starts to happen.

Because now, when we want to compute the gradient of the loss with respect to h zero, we need to backpropagate through every one of these RNN cells.

And every time you backpropagate through one cell, you'll pick up one of these w transpose factors.

So that means that the final expression for the gradient on h zero will involve many, many factors of this weight matrix, which could be kind of bad.

Maybe don't think about the weight, the matrix case, but imagine a scaler case.

If we end up, if we have some scaler and we multiply by that same number over and over and over again, maybe not for four examples, but for something like a hundred or several hundred-time steps, then multiplying by the same number over and over again is really bad.

In the scaler case, it's either going to explode in the case that that number is greater than one or it's going to vanish towards zero in the case that number is less than one in absolute value.

And the only way in which this will not happen is if that number is exactly one, which is actually very rare to happen in practice.

That leaves us to, that same intuition extends to the matrix case, but now, rather than the absolute value of a scaler number, you instead need to look at the largest, the largest singular value of this weight matrix.

Now if that largest singular value is greater than one, then during this backward pass, when we multiply by the weight matrix over and over, that gradient on h w, on h zero, sorry, will become very, very large, when that matrix is too large.

And that's something we call the exploding gradient problem.

Where now this gradient will explode exponentially in depth with the number of time steps that we backpropagate through.

And if the largest singular value is less than one, then we get the opposite problem, where now our gradients will shrink and shrink and shrink exponentially, as we backpropagate and pick up more and more factors of this weight matrix.

That's called the vanishing gradient problem.

There's a bit of a hack that people sometimes do to fix the exploding gradient problem called gradient clipping, which is just this simple heuristic saying that after we compute our gradient, if that gradient, if it's L2 norm is above some threshold, then just clamp it down and divide, just clamp it down so it has this maximum threshold.

This is kind of a nasty hack, but it actually gets used in practice quite a lot when training recurrent neural networks.

And it's a relatively useful tool for attacking this exploding gradient problem.

But now for the vanishing gradient problem, what we typically do is we might need to move to a more complicated RNN architecture.

So that motivates this idea of an LSTM.

An LSTM, which stands for Long Short Term Memory, is this slightly fancier recurrence relation for these recurrent neural networks.

It's really designed to help alleviate this problem of vanishing and exploding gradients.

So that rather than kind of hacking on top of it, we just kind of design the architecture to have better gradient flow properties.

Kind of an analogy to those fancier CNN architectures that we saw at the top of the lecture.

Another thing to point out is that the LSTM cell actually comes from 1997.

So this idea of an LSTM has been around for quite a while, and these folks were working on these ideas way back in the 90s, were definitely ahead of the curve.

Because these models are kind of used everywhere now 20 years later.

And LSTMs kind of have this funny functional form.

So remember when we had this vanilla recurrent neural network, it had this hidden state.

And we used this recurrence relation to update the hidden state at every time step.

Well, now in an LSTM, we actually have two, we maintain two hidden states at every time step.

One is this h t, which is called the hidden state, which is kind of an analogy to the hidden state that we had in the vanilla RNN.

But an LSTM also maintains the second vector, c t, called the cell state.

And the cell state is this vector which is kind of internal, kept inside the LSTM, and it does not really get exposed to the outside world.

And we'll see, and you can kind of see that through this update equation, where you can see that when we, first when we compute these, we take our two inputs, we use them to compute these four gates called i, f, o, n, g.

We use those gates to update our cell states, c t, and then we expose part of our cell state as the hidden state at the next time step.

This is kind of a funny functional form, and I want to walk through for a couple slides exactly why do we use this architecture and why does it make sense, especially in the context of vanishing or exploding gradients.

This first thing that we do in an LSTM is that we're given this previous hidden state, h t, and we're given our current input vector, x t, and just like the vanilla RNN.

In the vanilla RNN, remember, we took those two input vectors.

We concatenated them.

Then we did a matrix multiply to directly compute the next hidden state in the RNN.

Now, the LSTM does something a little bit different.

We're going to take our previous hidden state and our current input, stack them, and now multiply by a very big weight matrix, w, to compute four different gates, Which all have the same size as the hidden state.

Sometimes, you'll see this written in different ways.

Some authors will write a different weight matrix for each gate.

Some authors will combine them all into one big weight matrix.

But it's all really the same thing.

The ideas is that we take our hidden state, our current input, and then we use those to compute these four gates.

These four gates are the, you often see this written as i, f, o, g, ifog, which makes it pretty easy to remember what they are.

I is the input gate.

It says how much do we want to input into our cell.

F is the forget gate.

How much do we want to forget the cell memory at the previous, from the previous time step.

O is the output gate, which is how much do we want to reveal ourself to the outside world.

And G really doesn't have a nice name, so I usually call it the gate gate.

G, it tells us how much we want to write into our input cell.

And then you notice that each of these four gates are using a different non linearity.

The input, forget and output gate are all using sigmoids, which means that their values will be between zero and one.

Whereas the gate uses a tanh, which means it's output will be between minus one and one.

So, these are kind of weird, but it makes a little bit more sense if you imagine them all as binary values.

Right, like what happens at the extremes of these two values?

It's kind of what happens, if you look after we compute these gates if you look at this next equation, you can see that our cell state is being multiplied element wise by the forget gate.

Sorry, our cell state from the previous time step is being multiplied element wise by this forget gate.

And now if this forget gate, you can think of it as being a vector of zeros and ones, that's telling us for each element in the cell state, do we want to forget that element of the cell in the case if the forget gate was zero?

Or do we want to remember that element of the cell in the case if the forget gate was one.

Now, once we've used the forget gate to gate off the part of the cell state, then we have the second term, which is the element wise product of i and g.

So now, i is this vector of zeros and ones, cause it's coming through a sigmoid, telling us for each element of the cell state, do we want to write to that element of the cell state in the case that i is one, or do we not want to write to that element of the cell state at this time step in the case that i is zero.

And now the gate gate, because it's coming through a tanh, will be either one or minus one.

So that is the value that we want, the candidate value that we might consider writing to each element of the cell state at this time step.

Then if you look at the cell state equation, you can see that at every time step, the cell state has these kind of these different, independent scaler values, and they're all being incremented or decremented by one.

So there's kind of like, inside the cell state, we can either remember or forget our previous state, and then we can either increment or decrement each element of that cell state by up to one at each time step.

So you can kind of think of these elements of the cell state as being little scaler integer counters that can be incremented and decremented at each time step.

And now, after we've computed our cell state, then we use our now updated cell state to compute a hidden state, which we will reveal to the outside world.

So because this cell state has this interpretation of being counters, and sort of counting up by one or minus one at each time step, we want to squash that counter value into a nice zero to one range using a tanh.

And now, we multiply element wise, by this output gate.

And the output gate is again coming through a sigmoid, so you can think of it as being mostly zeros and ones, and the output gate tells us for each element of our cell state, do we want to reveal or not reveal that element of our cell state when we're computing the external hidden state for this time step.

And then, I think there's kind of a tradition in people trying to explain LSTMs, that everyone needs to come up with their own potentially confusing LSTM diagram.

So here's my attempt.

Here, we can see what's going on inside this LSTM cell, is that we take our, we're taking as input on the left our previous cell state and the previous hidden state, as well as our current input, x t.

Now we're going to take our current, our previous hidden state, as well as our current input, stack them, and then multiply with this weight matrix, w, to produce our four gates.

And here, I've left out the non-linearities because we saw those on a previous slide.

And now the forget gate multiplies element wise with the cell state.

The input and gate are multiplied element wise and added to the cell state.

And that gives us our next cell.

The next cell gets squashed through a tanh, and multiplied element wise with this output gate to produce our next hidden state.

Question?

No, So they're coming through this, they're coming from different parts of this weight matrix.

So, if our hidden, if our x and our h all have this dimension h, then after we stack them, they'll be a vector size two h, and now our weight matrix will be this matrix of size four h times two h.

So, you can think of that as sort of having four chunks of this weight matrix.

And each of these four chunks of the weight matrix is going to compute a different one of these gates.

You'll often see this written for clarity, kind of combining all four of those different weight matrices into a single large matrix, w, just for notational convenience.

But they're all computed using different parts of the weight matrix.

But you're correct in that they're all computed using the same functional form of just stacking the two things and taking the matrix multiplication.

Now that we have this picture, we can think about what happens to an LSTM cell during the backwards pass?

We saw, in the context of vanilla recurrent neural network, that some bad things happened during the backwards pass, where we were continually multiplying by that weight matrix, w.

But now, the situation looks much, quite a bit different in the LSTM.

If you imagine this path backwards of computing the gradients of the cell state, we get quite a nice picture.

Now, when we have our upstream gradient from the cell coming in, then once we backpropagate backwards through this addition operation, remember that this addition just copies that upstream gradient into the two branches, so our upstream gradient gets copied directly and passed directly to backpropagating through this element wise multiply.

So then our upstream gradient ends up getting multiplied element wise by the forget gate.

As we backpropagate backwards through this cell state, the only thing that happens to our upstream cell state gradient is that it ends up getting multiplied element wise by the forget gate.

This is really a lot nicer than the vanilla RNN for two reasons.

One is that this forget gate is now an element wise multiplication rather than a full matrix multiplication.

So element wise multiplication is going to be a little bit nicer than full matrix multiplication.

Second is that element wise multiplication will potentially be multiplying by a different forget gate at every time step.

So remember, in the vanilla RNN, we were continually multiplying by that same weight matrix over and over again, which led very explicitly to these exploding or vanishing gradients.

But now in the LSTM case, this forget gate can vary from each time step.

Now, it's much easier for the model to avoid these problems of exploding and vanishing gradients.

Finally, because this forget gate is coming out from a sigmoid, this element wise multiply is guaranteed to be between zero and one, which again, leads to sort of nicer numerical properties if you imagine multiplying by these things over and over again.

Another thing to notice is that in the context of the vanilla recurrent neural network, we saw that during the backward pass, our gradients were flowing through also a tanh at every time step.

But now, in an LSTM, our outputs are, in an LSTM, our hidden state is used to compute those outputs, y t, so now, each hidden state, if you imagine backpropagating from the final hidden state back to the first cell state, then through that backward path, we only backpropagate through a single tanh non-linearity rather than through a separate tanh at every time step.

So kind of when you put all these things together, you can see this backwards pass backpropagating through the cell state is kind of a gradient super highway that lets gradients pass relatively unimpeded from the loss at the very end of the model all the way back to the initial cell state at the beginning of the model.

Was there a question?

Yeah, what about the gradient in respect to w?

Cause that's ultimately the thing that we care about.

So, the gradient with respect to w will come through, at every time step, will take our current cell state as well as our current hidden state and that will give us an element, that will give us our local gradient on w for that time step.

So because our cell state, and just in the vanilla RNN case, we'll end up adding those first time step w gradients to compute our final gradient on w.

But now, if you imagine the situation where we have a very long sequence, and we're only getting gradients to the very end of the sequence.

Now, as you backpropagate through, we'll get a local gradient on w for each time step, and that local gradient on w will be coming through these gradients on c and h.

So because we're maintaining the gradients on c much more nicely in the LSTM case, those local gradients on w at each time step will also be carried forward and backward through time much more cleanly.

Another question?

Yeah, so the question is due to the non linearities, could this still be susceptible to vanishing gradients?

And that could be the case.

Actually, so one problem you might imagine is that maybe if these forget gates are always less than zero, or always less than one, you might get vanishing gradients as you continually go through these forget gates.

Well, one sort of trick that people do in practice is that they will, sometimes, initialize the biases of the forget gate to be somewhat positive.

So that at the beginning of training, those forget gates are always very close to one.

So that at least at the beginning of training, then we have not so, relatively clean gradient flow through these forget gates, since they're all initialized to be near one.

And then throughout the course of training, then the model can learn those biases and kind of learn to forget where it needs to.

You're right that there still could be some potential for vanishing gradients here.

But it's much less extreme than the vanilla RNN case, both because those fs can vary at each time step, and also because we're doing this element wise multiplication rather than a full matrix multiplication.

So you can see that this LSTM actually looks quite similar to ResNet.

In this residual network, we had this path of identity connections going backward through the network and that gave, sort of a gradient super highway for gradients to flow backward in ResNet.

And now it's kind of the same intuition in LSTM where these additive and element wise multiplicative interactions of the cell state can give a similar gradient super highway for gradients to flow backwards through the cell state in an LSTM.

And by the way, there's this other kind of nice paper called highway networks, which is kind of in between this idea of this LSTM cell and these residual networks.

So these highway networks actually came before residual networks, and they had this idea where at every layer of the highway network, we're going to compute sort of a candidate activation, as well as a gating function that tells us that interrelates between our previous input at that layer, and that candidate activation that came through our convolutions or what not.

So there's actually a lot of architectural similarities between these things, and people take a lot of inspiration from training very deep CNNs and very deep RNNs and there's a lot of crossover here.

Very briefly, you'll see a lot of other types of variance of recurrent neural network architectures out there in the wild.

Probably the most common, apart from the LSTM, is this GRU, called the gated recurrent unit.

And you can see those update equations here, and it kind of has this similar flavor of the LSTM, where it uses these multiplicative element wise gates together with these additive interactions to avoid this vanishing gradient problem.

There's also this cool paper called LSTM: a search based odyssey, very inventive title, where they tried to play around with the LSTM equations and swap out the non linearities at one point, like do we really need that tanh for exposing the output gate, and they tried to answer a lot of these different questions about each of those non linearities, each of those pieces of the LSTM update equations.

What happens if we change the model and tweak those LSTM equations a little bit.

And kind of the conclusion is that they all work about the same Some of them work a little bit better than others for one problem or another.

But generally, none of the things, none of the tweaks of LSTM that they tried were significantly better that the original LSTM for all problems.

So that gives you a little bit more faith that the LSTM update equations seem kind of magical but they're useful anyway.

You should probably consider them for your problem.

There's also this cool paper from Google a couple years ago where they tried to use, where they did kind of an evolutionary search and did a search over many, over a very large number of random RNN architectures, they kind of randomly permute these update equations and try putting the additions and the multiplications and the gates and the non-linearities in different kinds of combinations.

They blasted this out over their huge Google cluster and just tried a whole bunch of these different weigh updates in various flavors.

And again, it was the same story that they didn't really find anything that was significantly better than these existing GRU or LSTM styles.

Although there were some variations that worked maybe slightly better or worse for certain problems.

But kind of the take away is that probably and using an LSTM or GRU is not so much magic in those equations, but this idea of managing gradient flow properly through these additive connections and these multiplicative gates is super useful.

So yeah, the summary is that RNNs are super cool.

They can allow you to attack tons of new types of problems.

They sometimes are susceptible to vanishing or exploding gradients.

But we can address that with weight clipping and with fancier architectures.

And there's a lot of cool overlap between CNN architectures and RNN architectures.

So next time, you'll be taking the midterm.

But after that, we'll have a, sorry, a question?

Midterm is after this lecture so anything up to this point is fair game.

And so you guys, good luck on the midterm on Tuesday.