Course 5: Sequence Models

Notebook:	Programming & other Learning		
Created:	19/4/2018 5:41 PM	**Updated:**	24/5/2018 3:06 PM
Author:	Pradeep Pant		

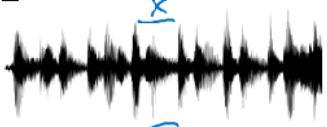
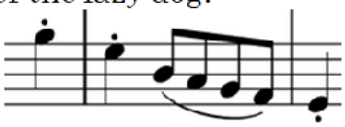
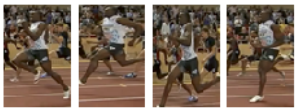
****
Course 5: Sequence Models
****
In this course we'll learn how to build models for natural language, audio, and other sequence data. Thanks to deep learning, sequence algorithms are working far better than just two years ago, and this is enabling numerous exciting applications in speech recognition, music synthesis, chatbots, machine translation, natural language understanding, and many others.

You will:
- Understand how to build and train Recurrent Neural Networks (RNNs), and commonly-used variants such as GRUs and LSTMs.
- Be able to apply sequence models to natural language problems, including text synthesis.
- Be able to apply sequence models to audio applications, including speech recognition and music synthesis.

Week 1: Foundations of Convolutional Neural Networks
Learn about recurrent neural networks. This type of model has been proven to perform extremely well on temporal data. It has several variants including LSTMs, GRUs and Bidirectional RNNs, which you are going to learn about in this section.

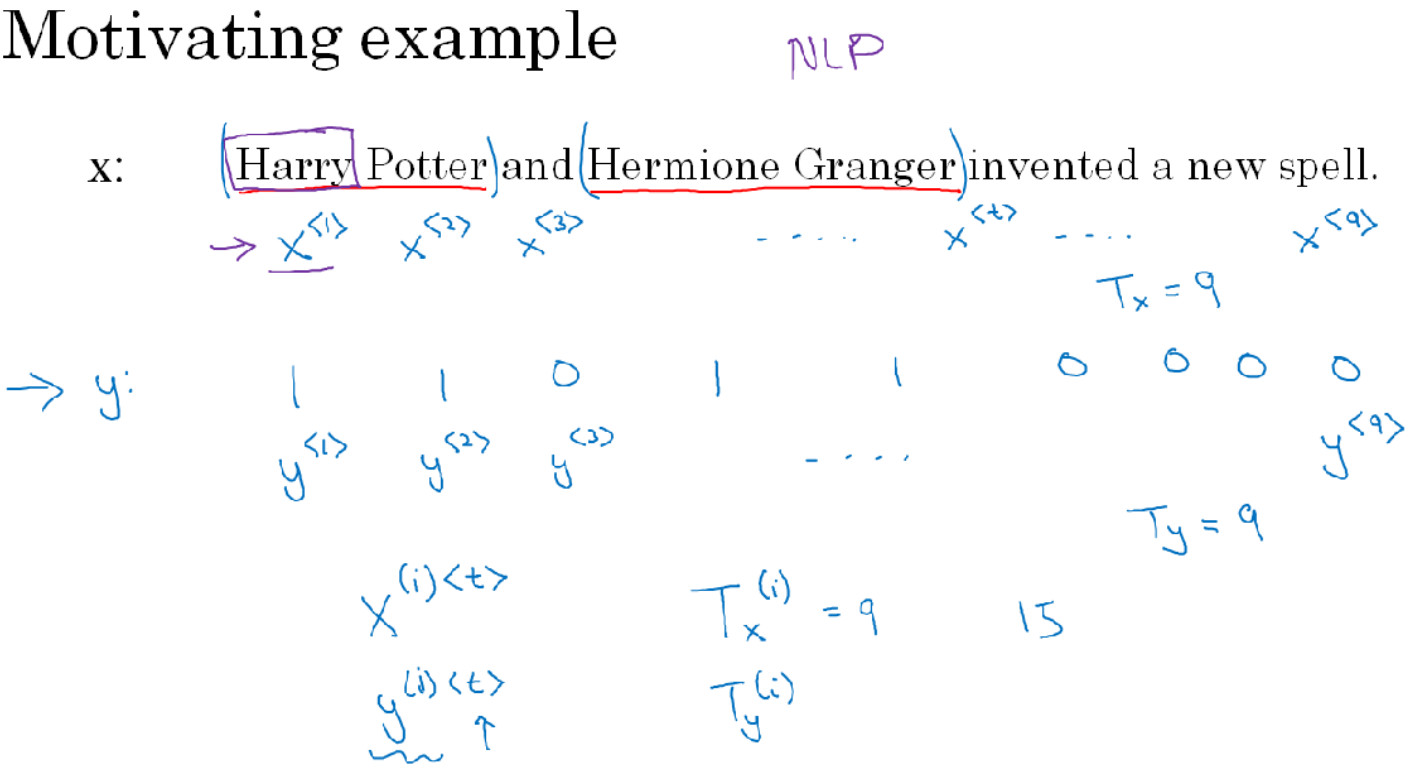
Recurrent Neural Networks
Why sequence models
In this course, we learn about sequence models, one of the most exciting areas in deep learning. Models like recurrent neural networks or RNNs have transformed speech recognition, natural language processing and other areas and in this course, we'll learn how to build these models for yourself. Let's start by looking at a few examples of where sequence models can be useful. In speech recognition you are given an input audio clip X and asked to map it to a text transcript Y. Both the input and the output here are sequence data, because X is an audio clip and so that plays out over time and Y, the output, is a sequence of words. So sequence models such as a recurrent neural networks and other variations, we'll learn about in a little bit have been very useful for speech recognition. Music generation is another example of a problem with sequence data. In this case, only the output Y is a sequence, the input can be the empty set, or it can be a single integer, maybe referring to the genre of music you want to generate or maybe the first few notes of the piece of music you want. But here X can be nothing or maybe just an integer and output Y is a sequence. In sentiment classification the input X is a sequence, so given the input phrase like, "There is nothing to like in this movie" how many stars do you think this review will be? Sequence models are also very useful for DNA sequence analysis. So your DNA is represented via the four alphabets A, C, G, and T. And so given a DNA sequence can you label which part of this DNA sequence say corresponds to a protein. In machine translation you are given an input sentence, voulez-vou chante avec moi? And you're asked to output the translation in a different language. In video activity recognition you might be given a sequence of video frames and asked to recognize the activity and in name entity recognition you might be given a sentence and asked to identify the people in that sentence. So all of these problems can be addressed as supervised learning with label data X, Y as the training set. But, as you can tell from this list of examples, there are a lot of different types of sequence problems. In some, both the input X and the output Y are sequences, and in that case, sometimes X and Y can have different lengths, or in this example and this example, X and Y have the same length and in some of these examples only either X or only the opposite Y is a sequence. So in this course we'll learn about sequence models are applicable, so all of these different settings. See diagram below of some of the examples of sequence data.

Examples of sequence data

Speech recognition		"The quick brown fox jumped over the lazy dog."
Music generation		
Sentiment classification	"There is nothing to like in this movie."	★☆☆☆☆
DNA sequence analysis	AGCCCCTGTGAGGAACTAG	AGCCCCTGTCAGGAACTAG
Machine translation	Voulez-vous chanter avec moi?	Do you want to sing with me?
Video activity recognition		Running
Name entity recognition	Yesterday, Harry Potter met Hermione Granger.	Yesterday, Harry Potter met Hermione Granger.

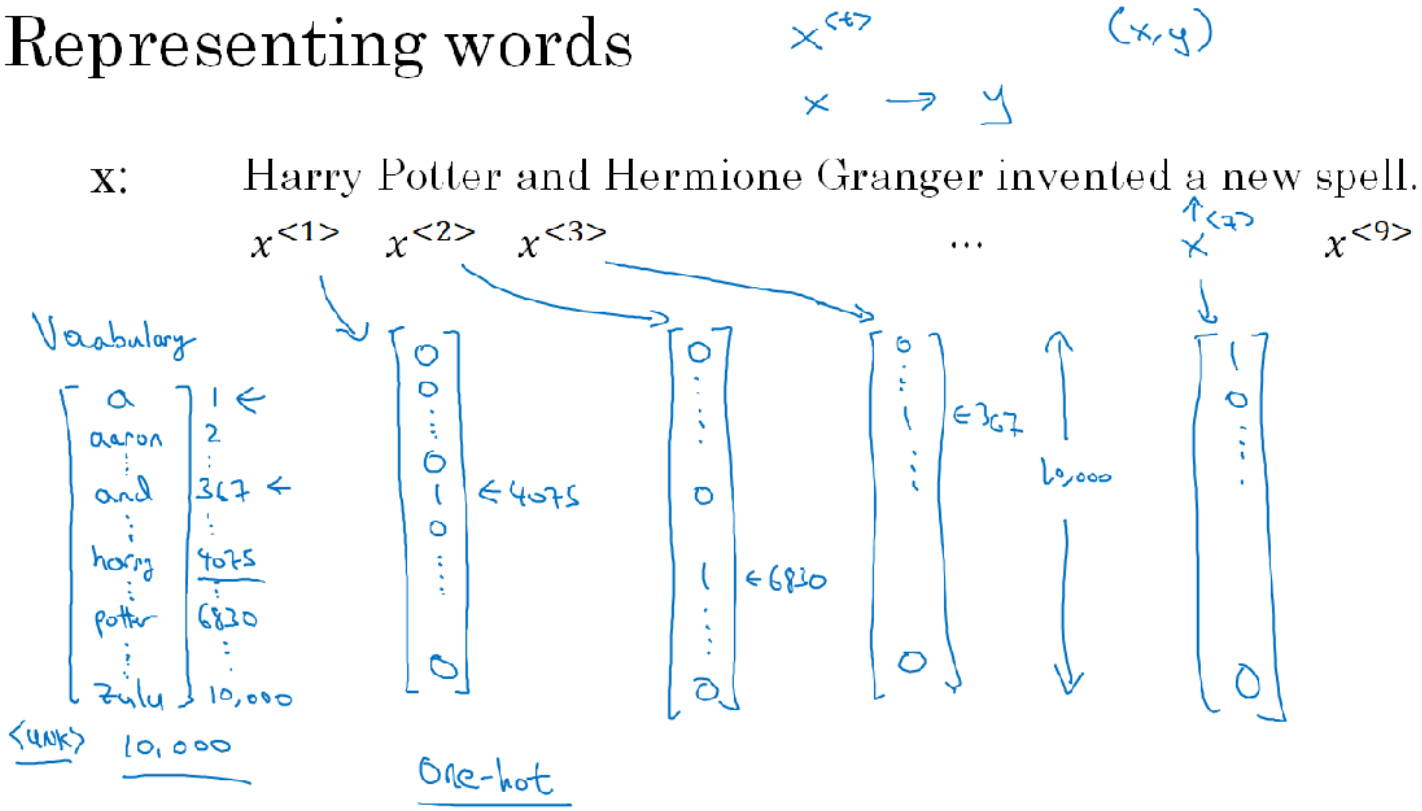
Notation

In the last section, we saw some of the wide range of applications through which you can apply sequence models. Let's start by defining a notation that we'll use to build up these sequence models. As a motivating example, let's say you want to build a sequence model to input a sentence like this, Harry Potter and Hermione Granger invented a new spell and these are characters by the way, from the Harry Potter sequence of novels by J. K. Rowling. And let say you want a sequence model to automatically tell you where are the peoples names in this sentence. So, this is a problem called **Named-entity recognition** and this is used by search engines for example, to index all of say the last 24 hours news of all the people mentioned in the news articles so that they can index them appropriately and name into the recognition systems can be used to find people's names, companies names, times, locations, countries names, currency names, and so on in different types of text. Now, given this input x let's say that you want a model to operate y that has one outputs per input word and the target output the design y tells you for each of the input words is that part of a person's name and technically this maybe isn't the best output representation, there are some more sophisticated output representations that tells you not just is a word part of a person's name, but tells you where are the start and ends of people's names their sentence



you want to know Harry Potter starts here, and ends here, starts here, and ends here. But for this motivating example, I'm just going to stick with this simpler output representation. Now, the input is the sequence of nine words. So, eventually we're going to have nine sets of features to represent these nine words, and index into the positions and sequence, I'm going to use X and then superscript angle brackets 1, 2, 3 and so on up to X angle brackets nine to index into the different positions. I'm going to use $X^{<t>}$ with the index t to index into positions, in the middle of the sequence and t implies that these are **temporal sequences** although whether the sequences are temporal one or not, I'm going to use the index t to index into the positions in the sequence and similarly for the outputs, we're going to refer to these outputs as y and go back at 1, 2, 3 and so on up to y nine. Let's also used T_X to denote the length of the input sequence, so in this case there are nine words. So T_x is equal to 9 and we used Ty to denote the length of the output sequence. In this example T_x is equal to Ty but you saw on the last section T_x and T_y can be different. So, you will remember that in the notation we've been using, we've been writing X round brackets i to denote the i training example. So, to refer to the t^{th} element or the t^{th} element in the sequence of training example i will use this notation and if Tx is the length of a sequence then different examples in your training set can have different lengths. And so $T_x^{(i)}$ would be the input sequence length for training example i, and similarly $y^{(i)<t>}$ means the t^{th} element in the output sequence of the i for an example and $Ty^{(i)}$ will be the length of the output sequence in the i training example. So into this example, $T_x^{(i)}$ is equal to 9 would be the highly different training example with a sentence of 15 words and $T_x^{(i)}$ will be close to 15 for that different training example. Now, that we're starting to work in **NLP or Natural Language Processing**. **Now, this is our first serious foray into NLP or Natural Language Processing** and one of the things we need to decide is, how to represent individual words in the sequence. So, how do you represent a word like Harry, and why should $x^{<1>}$ really be? Let's next talk about how we would represent individual words in a sentence.

Representing words



So, to represent a word in the sentence the first thing you do is come up with a Vocabulary. Sometimes also called a **Dictionary** and that means making a list of the words that you will use in your representations. So the first word in the vocabulary is a, that will be the first word in the dictionary. The second word is Aaron and then a little bit further down is the word and, and then eventually you get to the words Harry then eventually the word Potter, and then all the way down to maybe the last word in dictionary is Zulu. And so, a will be word one, Aaron is word two, and in my dictionary the word and appears in positional index 367. Harry appears in position 4075, Potter in position 6830, and Zulu is the last word to the dictionary is maybe word 10,000.

So in this example, I'm going to use a dictionary with size 10,000 words. This is quite small by modern NLP applications. For commercial applications, for visual size commercial applications, dictionary sizes of 30 to 50,000 are more common and 100,000 is not uncommon and then some of the large Internet companies will use dictionary sizes that are maybe a million words or even bigger than that. But you see a lot of commercial applications used dictionary sizes of maybe 30,000 or maybe 50,000 words. But I'm going to use 10,000 for illustration since it's a nice round number. So, if you have chosen a dictionary of 10,000 words and one way to build this dictionary will be be to look through your training sets and find the top 10,000 occurring words, also look through some of the online dictionaries that tells you what are the most common 10,000 words in the English Language saved. What you can do is then use one hot representations to represent each of these words. For example, $x^{<1>}$ which represents the word Harry would be a vector with all zeros except for a 1 in position 4075 because that was the position of Harry in the dictionary and then $x^{<2>}$ will be again similarly a vector of all zeros except for a 1 in position 6830 and then zeros everywhere else. The word and was represented as position 367 so $x^{<3>}$ would be a vector with zeros of 1 in position 367 and then zeros everywhere else and each of these would be a 10,000 dimensional vector if your vocabulary has 10,000 words and the word "a" is the first word of the dictionary, then $x^{<7>}$ which corresponds to word a, that would be the vector 1. This is the first element of the dictionary and then zero everywhere else. So in this representation, $x^{<t>}$ for each of the values of t in a sentence will be a **one-hot vector**, one-hot because there's exactly one one is on and zero everywhere else and you will have nine of them to represent the nine words in this sentence and the goal is given this representation for X to learn a mapping using a sequence model to then target output y, I will do this as a supervised learning problem, I'm sure given the table data with both x and y.

Representing words

x: Harry Potter and Hermione Granger invented a new spell.

$x^{<1>}$ $x^{<2>}$ $x^{<3>}$ ... $x^{<9>}$

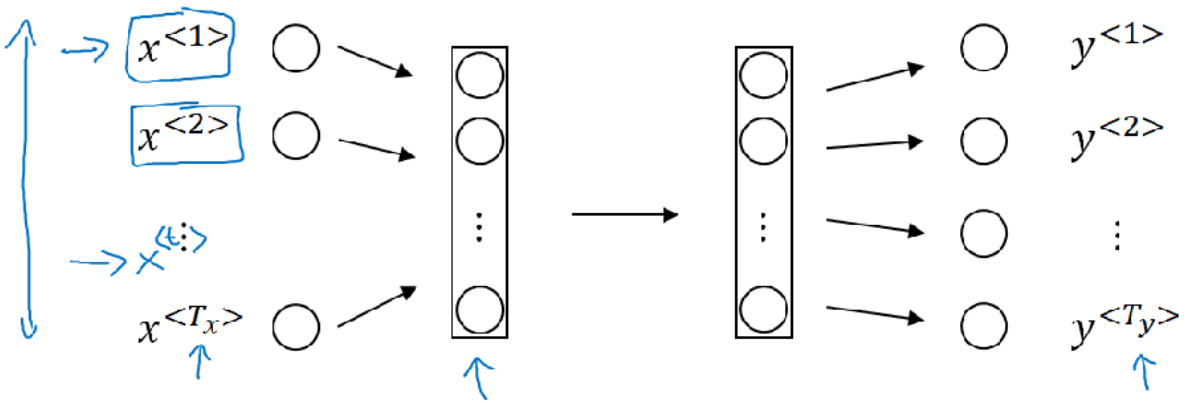
And = 367
Invented = 4700
A = 1
New = 5976
Spell = 8376
Harry = 4075
Potter = 6830
Hermione = 4200
Gran... = 4000

Then just one last detail, which we'll talk more about in a later section is, what if you encounter a word that is not in your vocabulary? Well the answer is, you create a new token or a new fake word called Unknown Word which under note as follows and go back as **UNK** to represent words not in your vocabulary, we'll come more to talk more about this later. So, to summarize in this section, we described a notation for describing your training set for both x and y for sequence data. In the next section let's start to describe a Recurrent Neural Networks for learning the mapping from X to Y.

Recurrent Neural Networks Model

In the last section, we saw the notation we used to define sequence learning problems. Now, let's talk about how you can build a model, build a neural network to drawing the mapping from X to Y. Now, one thing you could do is try to use a standard neural network for this task. So in our previous example, we had nine input words. So you could imagine trying to take these nine input words, maybe the nine one hot vectors and feeding them into a standard neural network, maybe a few hidden layers and then eventually, have this output the nine values zero or one that tell you whether each word is part of a person's name. But this turns out not to work well, and there are really two main problems with this. The first is that the inputs and outputs can be different lengths in different examples. So it's not as if every single example has the same input length T_x or the same output length T_y and maybe if every sentence had a maximum length, maybe you could pad, or zero pad every input up to that maximum length, but this still doesn't seem like a good representation and in a second, it might be more serious problem is that a naive neural network architecture like this below

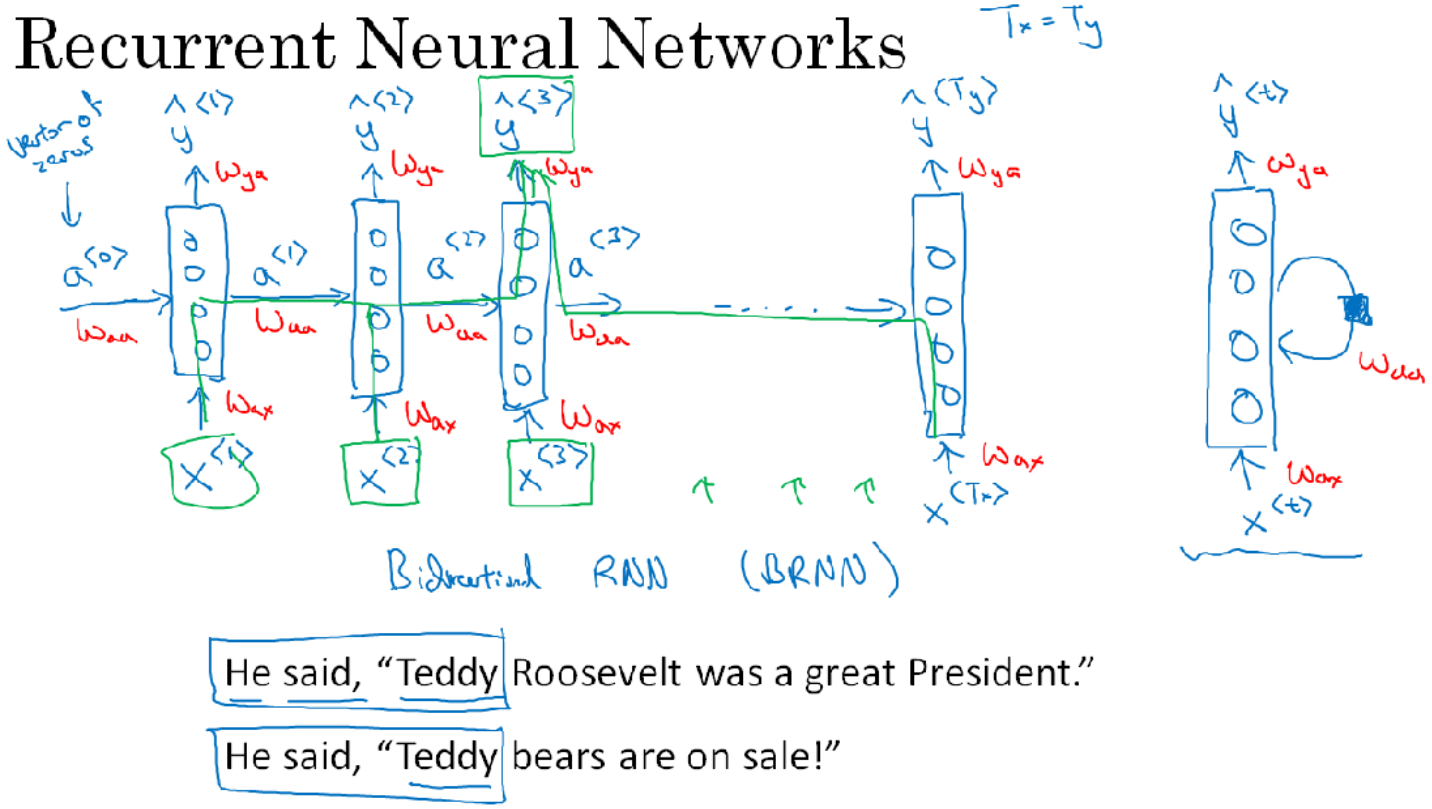
Why not a standard network?



Problems:

- Inputs, outputs can be different lengths in different examples.
- Doesn't share features learned across different positions of text.

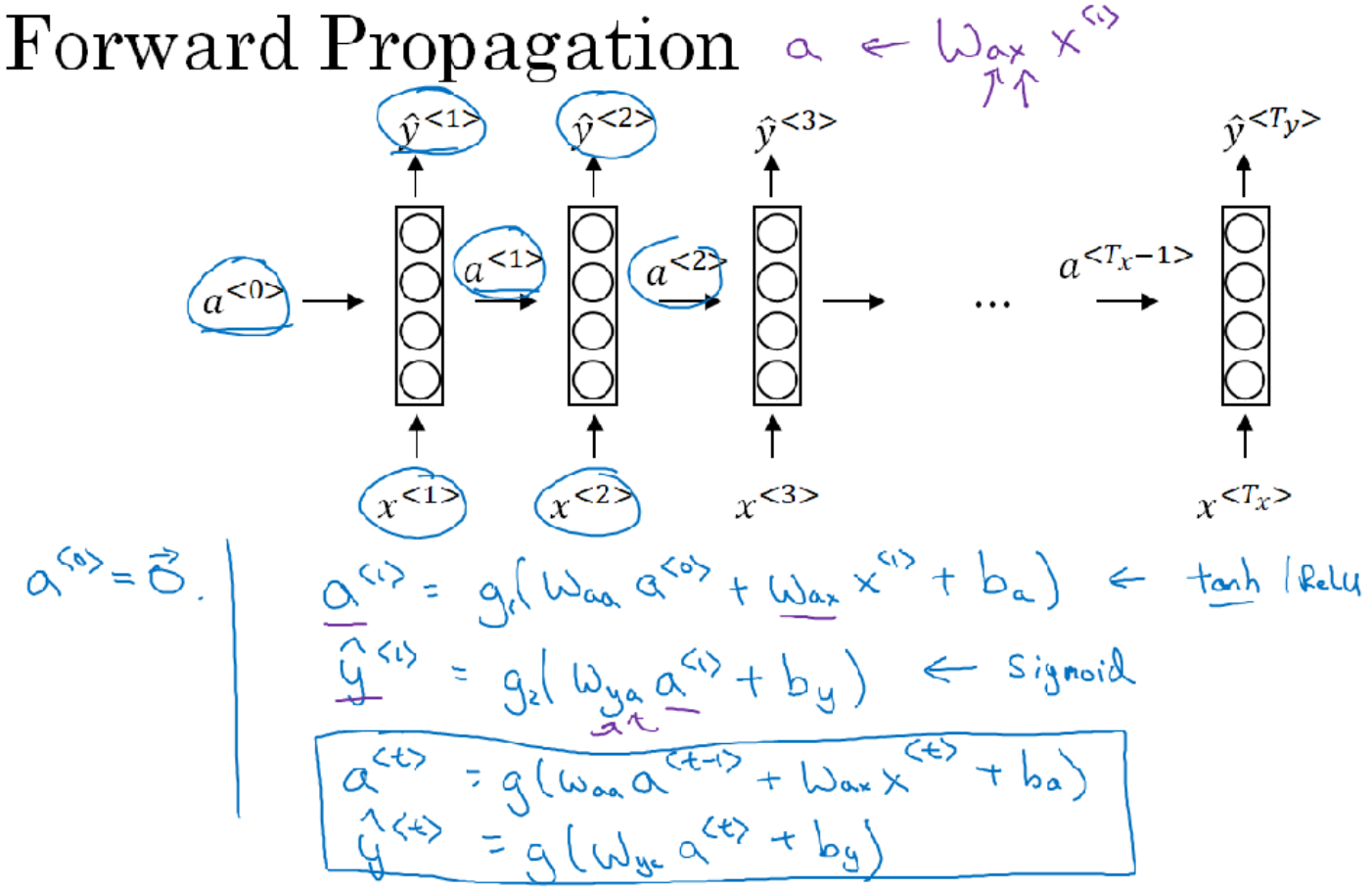
it doesn't share features learned across different positions of techs. In particular, if the neural network has learned that maybe the word heavy appearing in position one gives a sign that that is part of a person's name, then one would be nice if it automatically figures out that heavy appearing in some other position, XT also means that that might be a person's name. And this is maybe similar to what you saw in convolutional neural networks where you want things learned for one part of the image to generalize quickly to other parts of the image, and we'd like similar effect for sequence data as well. And similar to what you saw with convnets using a better representation will also let you reduce the number of parameters in your model.



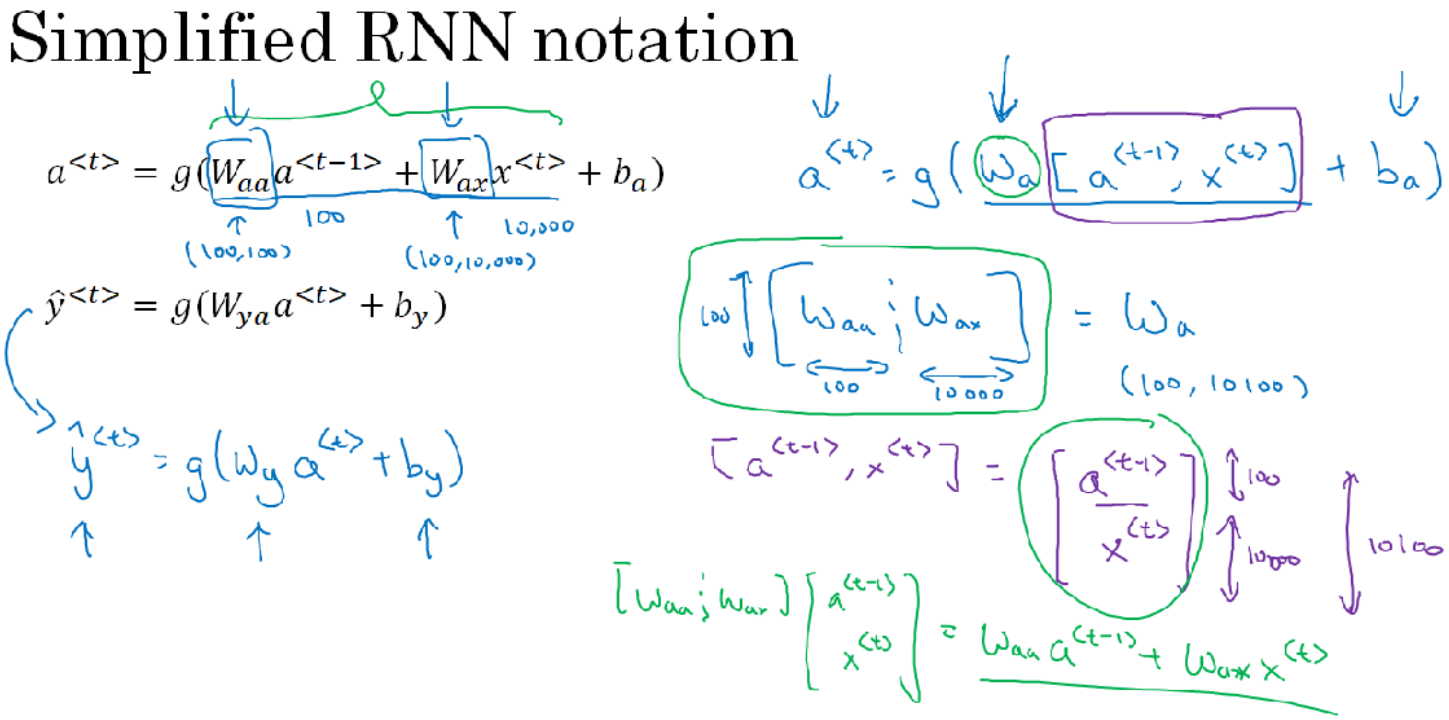
So previously, we said that each of these is a 10,000 dimensional one vector. And so, this is just a very large input layer. If the total input size was maximum number of words times 10,000, and the weight matrix of this first layer would end up having an enormous number of parameters. So a recurrent neural network which will start to describe in the next slide, does not have either of these disadvantages. So what is a recurrent neural network? Let's build one out. So if you are reading the sentence from left to right, the first word you read is the some first where say X1. What we're going to do is take the first word and feed it into a neural network layer. I'm going to draw it like this. So that's a hidden layer of the first neural network. And look at how the neural network maybe try to predict the output. So is this part of a person's name or not? And what a **recurrent neural network does is when it then goes on to read the second word in a sentence, say X2, instead of just predicting Y2 using only X2, it also gets to input some information from whether a computer that time-step ones. So in particular, the activation value from time-step one is passed on to time-step 2. And then, at the next time-step, a recurrent neural network inputs the third word X3, and it tries to predict, output some prediction y-hat 3, and so on, up until the last time-step where inputs XTx, and then it outputs Y hat TY.** In this example, Tx=Ty, and the architecture will change a bit if Tx and Ty are not identical. **And so, at each time-step, the recurrent neural network passes on this activation to the next time-step for it to use. And to kick off the whole thing, we'll also have some made up activation at time zero. This is usually the vector of zeroes.** Some researchers will initialize a zero randomly have other ways to initialize a zero but really having a vector zero is just a fake. Time Zero activation is the most common choice. And so that does input into the neural network. In some research papers or in some books, you see this type of neural network drawn with the following diagram in which every time-step, you input X and output Y hat, maybe sometimes there will be a T index there, and then to denote the recurrent connection, sometimes people will draw a loop like that, that the layer feeds back to itself. Sometimes they'll draw a shaded box to denote that this is the shaded box here, denotes a time delay of one step. I personally find these recurrent diagrams much harder to interpret. And so throughout this course, I will tend to draw the on the road diagram like the one you have on the left. But if you see something like the diagram on the right in a textbook or in a research paper, what it really means, or the way I tend to think about it is the mentally unrolled into the diagram you have on the left hand side. The recurrent neural network scans through the data from left to right. And the parameters it uses for each time step are shared. So there will be a set of parameters which we'll describe in greater detail on the next section, but the parameters governing

the connection from X1 to the hidden layer will be some set of the parameters we're going to write as WAX, and it's the same parameters WAX that it uses for every time-step I guess you could write WAX there as well. And the activations, the horizontal connections, will be governed by some set of parameters WAA, and is the same parameters WAA use on every time-step, and similarly, the sum WYA that governs the output predictions. And I'll describe in the next slide exactly how these parameters work. So in this recurrent neural network, what this means is that we're making the prediction for Y3 against the information not only from X3, but also the information from X1 and X2, because the information of X1 can pass through this way to help the prediction with Y3. Now one weakness of this RNN is that it only uses the information that is earlier in the sequence to make a prediction, in particular, when predicting Y3, it doesn't use information about the words X4, X5, X6 and so on. And so this is a problem because if you're given a sentence, he said, "Teddy Roosevelt was a great president." In order to decide whether or not the word Teddy is part of a person's name, it be really useful to know not just information from the first two words but to know information from the later words in the sentence as well, because the sentence could also happen, he said, "Teddy bears are on sale!" And so, given just the first three words, it's not possible to know for sure whether the word Teddy is part of a person's name. In the first example, it is, in the second example, is not, but you can't tell the difference if you look only at the first three words. So one limitation of this particular neural network structure is that the prediction at a certain time uses inputs or uses information from the inputs earlier in the sequence but not information later in the sequence. We will address this in a later video where we talk about a bidirectional recurrent neural networks or BRNNs. But for now, this simpler uni-directional neural network architecture will suffice for us to explain the key concepts. And we just have to make a quick modifications in these ideas later to enable say the prediction of Y-hat 3 to use both information earlier in the sequence as well as information later in the sequence, but we'll get to that in a later video. So let's not write to explicitly what are the calculations that this neural network does. Here's a cleaned out version of the picture of the neural network. As I mentioned previously, typically, you started off with the input a0 equals the vector of all zeroes.

Next. This is what a forward propagation looks like.



To compute a1, you would compute that as an activation function g, applied to Waa times a0 plus W a x times x1 plus a bias was going to write it as ba, and then to compute y hat 1 the prediction of times that one, that will be some activation function, maybe a different activation function, than the one above. But apply to WYA times a1 plus b y. And the notation convention I'm going to use for the sub zero of these matrices like that example, W a x. The second index means that this W a x is going to be multiplied by some x like quantity, and this means that this is used to compute some a like quantity. Like like so. And similarly, you notice that here WYA is multiplied by a sum a like quantity to compute a y type quantity. The activation function used in- to compute the activations will often be a tonnage and the choice of an RNN and sometimes, values are also used although the tonnage is actually a pretty common choice. And we have other ways of preventing the vanishing gradient problem which we'll talk about later this week. And depending on what your output y is, if it is a binary classification problem, then I guess you would use a sigmoid activation function or it could be a soft Max if you have a ky classification problem. But the choice of activation function here would depend on what type of output y you have. So, for the name entity recognition task, where Y was either zero or one. I guess the second g could be a signal and activation function. And I guess you could write g2 if you want to distinguish that this is these could be different activation functions but I usually won't do that. And then, more generally at time t, a t will be g of W a a times a, from the previous time-step, plus W a x of x from the current time-step plus B a, and y hat t is equal to g, again, it could be different activation functions but g of WYA times a t plus B y. So, these equations define for propagation in the neural network. Where you would start off with a zeroes [inaudible] and then using a zero and X1, you will compute a1 and y hat one, and then you, take X2 and use X2 and A1 to compute A2 and Y hat two and so on, and you carry out for propagation going from the left to the right of this picture. Now, in order to help us develop the more complex neural networks, I'm actually going to take this notation and simplify it a little bit.

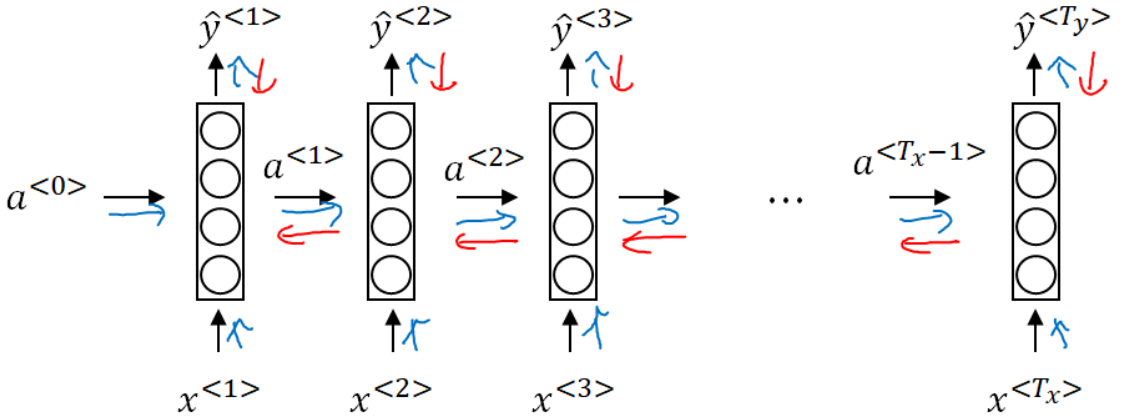


Backpropagation through time

We've already learned about the basic structure of an RNN. In this section, you'll see how backpropagation in a recurrent neural network works. As usual, when you implement this in one of the programming frameworks, often, the programming framework will automatically take care of backpropagation. But I think it's still useful to have a rough sense of how backprop works in RNNs. Let's take a look. You've seen how, for forward prop, you would computes these activations from left to right as follows in the neural network, and so you've outputs all of the predictions. In backprop, as you might already have guessed, you end up carrying backpropagation calculations in basically the

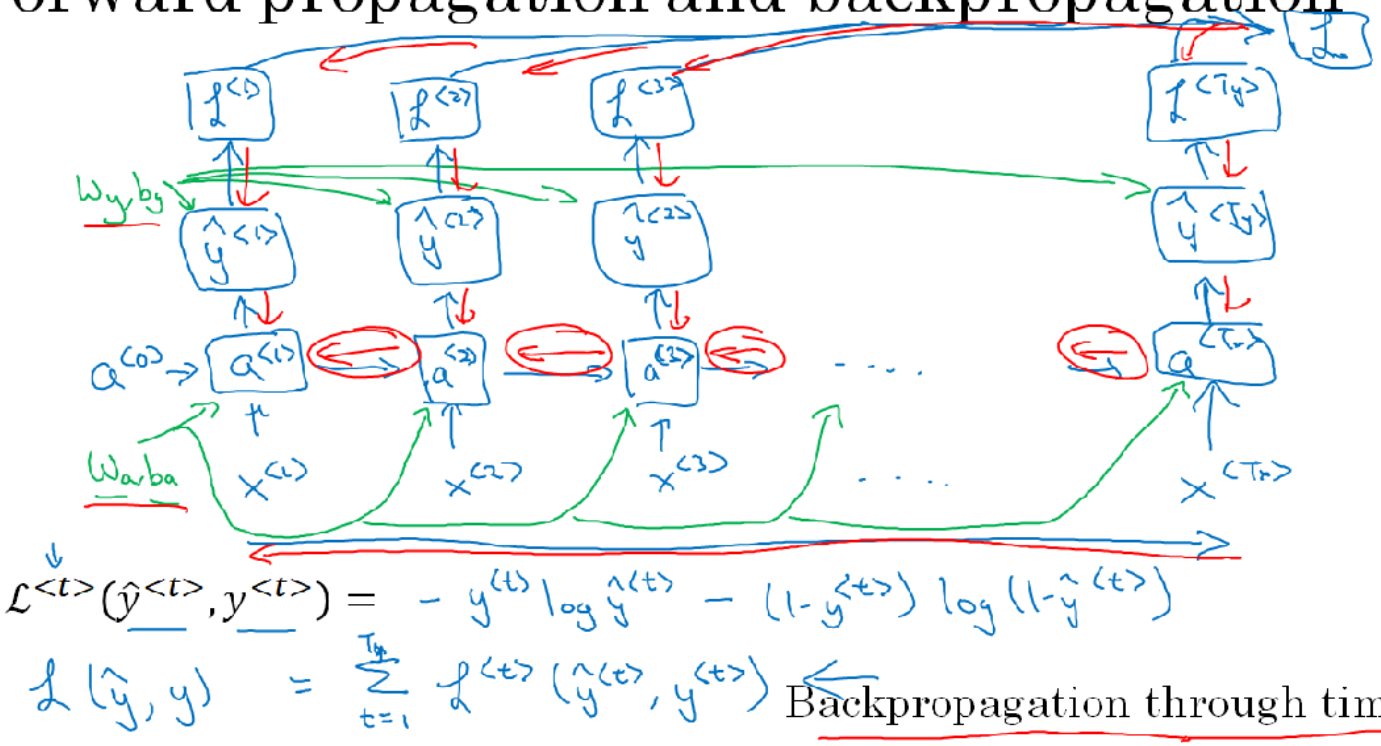
opposite direction of the forward prop arrows. So, let's go through the forward propagation calculation. You're given this input sequence x_1, x_2, x_3, up to x_tx. And then using x_1 and say, a_0, you're going to compute the activation, times that one, and then together, x_2 together with a_1 are used to compute a_2, and then a_3, and so on, up to a_tx. All right. And then to actually compute a_1, you also need the parameters.

Forward propagation and backpropagation



We'll just draw this in green, W_a and b_a, those are the parameters that are used to compute a_1. And then, these parameters are actually used for every single timestep so, these parameters are actually used to compute a_2, a_3, and so on, all the activations up to last timestep depend on the parameters W_a and b_a. Let's keep fleshing out this graph. Now, given a_1, your neural network can then compute the first prediction, y-hat_1, and then the second timestep, y-hat_2, y-hat_3, and so on, with y-hat_ty. And let me again draw the parameters of a different color. So, to compute y-hat, you need the parameters, W_y as well as b_y, and this goes into this node as well as all the others. So, I'll draw this in green as well. Next, in order to compute backpropagation, you need a loss function.

Forward propagation and backpropagation

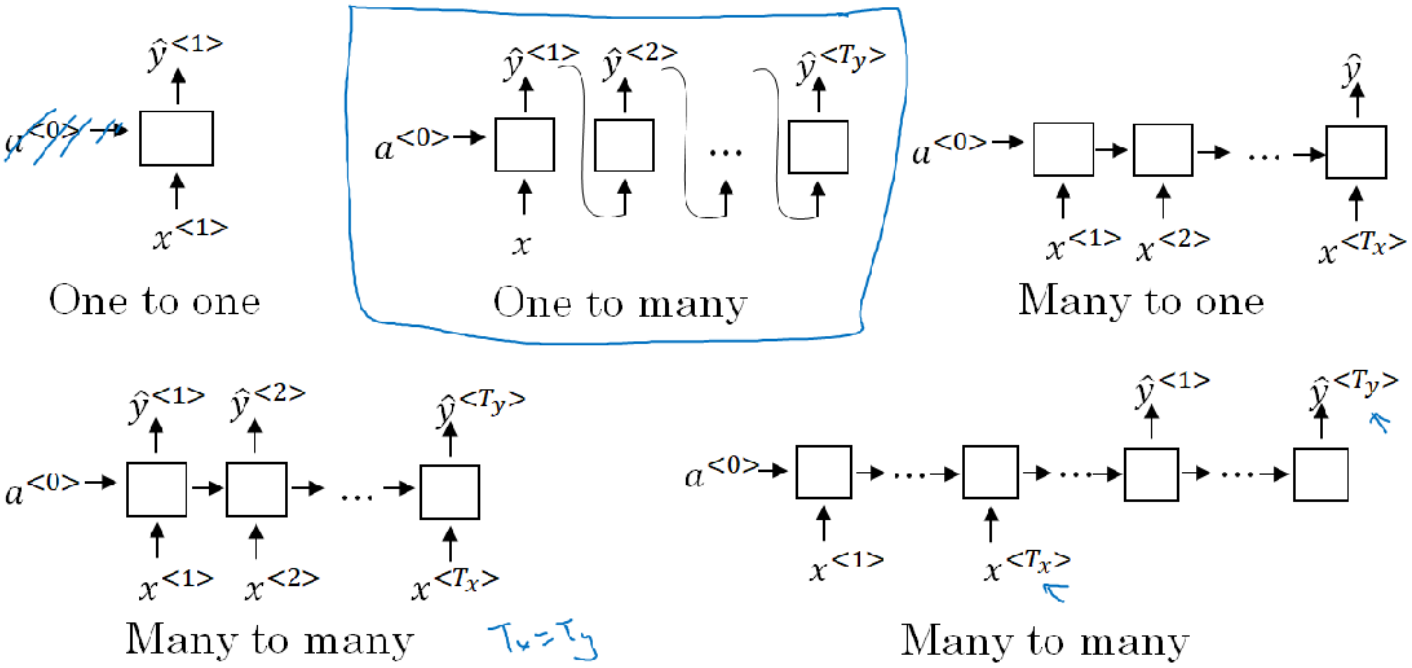


So let's define an element-wise loss force, which is supposed for a certain word in the sequence. It is a person's name, so y_t is one. And your neural network outputs some probability of maybe 0.1 of the particular word being a person's name. So I'm going to define this as the standard logistic regression loss, also called the cross entropy loss. This may look familiar to you from where we were previously looking at binary classification problems. So this is the loss associated with a single prediction at a single position or at a single time set, t, for a single word. Let's now define the overall loss of the entire sequence, so L will be defined as the sum overall t equals one to, i guess, T_x or T_y. T_x is equals to T_y in this example of the losses for the individual timesteps, comma y_t. And then, so, just have to L without this superscript T. This is the loss for the entire sequence. So, in a computation graph, to compute the loss given y-hat_1, you can then compute the loss for the first timestep given that you compute the loss for the second timestep, the loss for the third timestep, and so on, the loss for the final timestep. And then lastly, to compute the overall loss, we will take these and sum them all up to compute the final L using that equation, which is the sum of the individual per timestep losses. So, this is the computation problem and from the earlier examples **you've seen of backpropagation, it shouldn't surprise you that backprop then just requires doing computations or parsing messages in the opposite directions**. So, all of the four propagation steps arrows, so you end up doing that. And that then, allows you to compute all the appropriate quantities that lets you then, take the riveters, respected parameters, and update the parameters using gradient descent. **Now, in this back propagation procedure, the most significant message or the most significant recursive calculation is this one, which goes from right to left, and that's why it gives this algorithm as well, a pretty fast full name called backpropagation through time. And the motivation for this name is that for forward prop, you are scanning from left to right, increasing indices of the time, t, whereas, the backpropagation, you're going from right to left, you're kind of going backwards in time. So this gives this, I think a really cool name, backpropagation through time, where you're going backwards in time, right? That phrase really makes it sound like you need a time machine to implement this output, but I just thought that backprop through time is just one of the coolest names for an algorithm**. So, I hope that gives you a sense of how forward prop and backprop in RNN works. Now, so far, you've only seen this main motivating example in RNN, in which the length of the input sequence was equal to the length of the output sequence. In the next section, I want to show you a much wider range of RNN architecture, so I'll let you tackle a much wider set of applications.

Different types of RNNs

So far, you've seen an RNN architecture where the number of inputs, Tx, is equal to the number of outputs, Ty. It turns out that for other applications, Tx and Ty may not always be the same, and in this section, we'll see a much richer family of RNN architectures. You might remember from the first section of this week, where the input x and the output y can be many different types and it's not always the case that Tx has to be equal to Ty. In particular, in this example, Tx can be length one or even an empty set and then, an example like movie sentiment classification, the output y could be just an integer from 1 to 5, whereas the input is a sequence and in name entity recognition, in the example we're using, the input length and the output length are identical, but there are also some problems were the input length and the output length can be different.They're both our sequences but have different lengths, such as machine translation where a French sentence and English sentence can mean two different numbers of words to say the same thing. So it turns out that we could modify the basic RNN architecture to address all of these problems and the presentation in this section was inspired by a blog post by Andrej Karpathy, titled, **"The Unreasonable Effectiveness of Recurrent Neural Networks"**.

Summary of RNN types



Language model and sequence generation

Language modeling is one of the most basic and important tasks in natural language processing. There's also one that RNNs do very well. Where you build a language model and use it to generate Shakespeare-like texting, other types of text. Let's get started. So what is a language model? Let's say you're building this speech recognition system and you hear the sentence, the apple and pear salad was delicious. So what did you just hear me say? Did I say the apple and pair salad, or did I say the apple and pear salad?

You probably think the second sentence is much more likely, and in fact, that's what a good speech recognition system would help with even though these two sentences sound exactly the same and the way a speech recognition system picks the second sentence is by using a language model which tells it what the probability is of either of these two sentences.

What is language modelling?

Speech recognition

The apple and pair salad.

→ The apple and pear salad.

$P(\text{The apple and pair salad}) = 3.2 \times 10^{-13}$

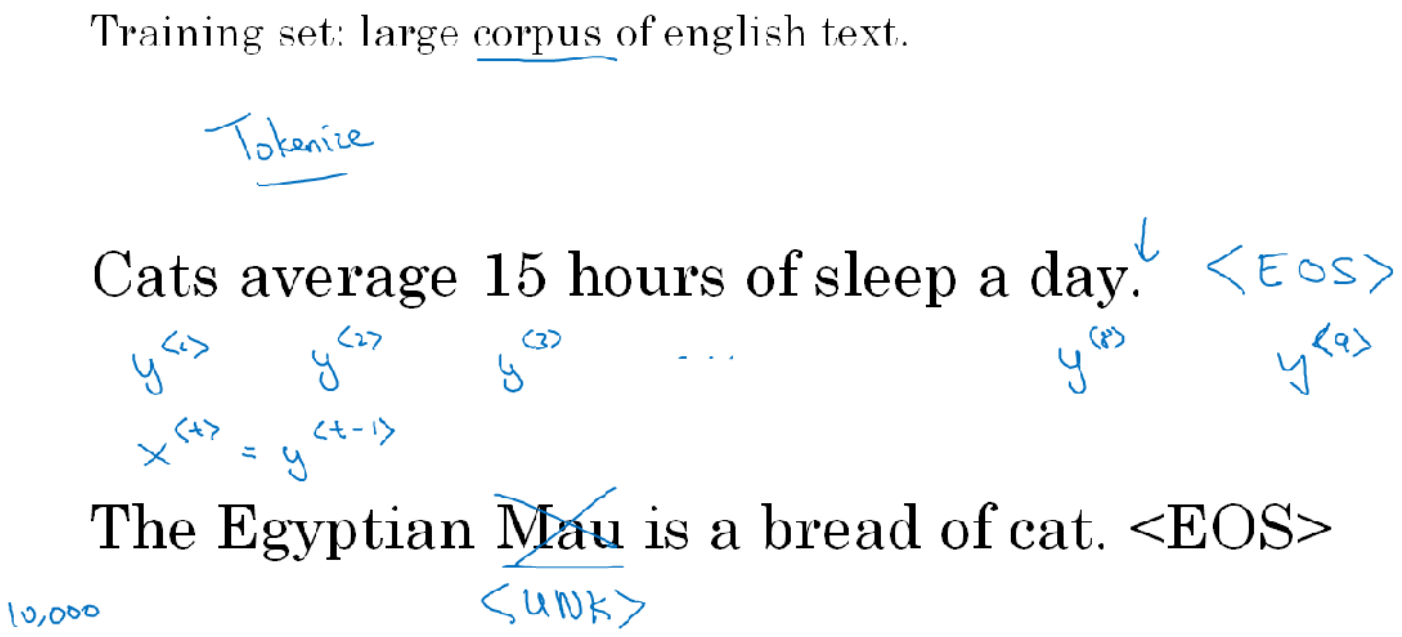
$P(\text{The apple and pear salad}) = 5.7 \times 10^{-10}$

$P(\text{Sentence}) = ?$ $\quad P(y^{<1>}, y^{<2>}, \ldots, y^{<T_y>})$

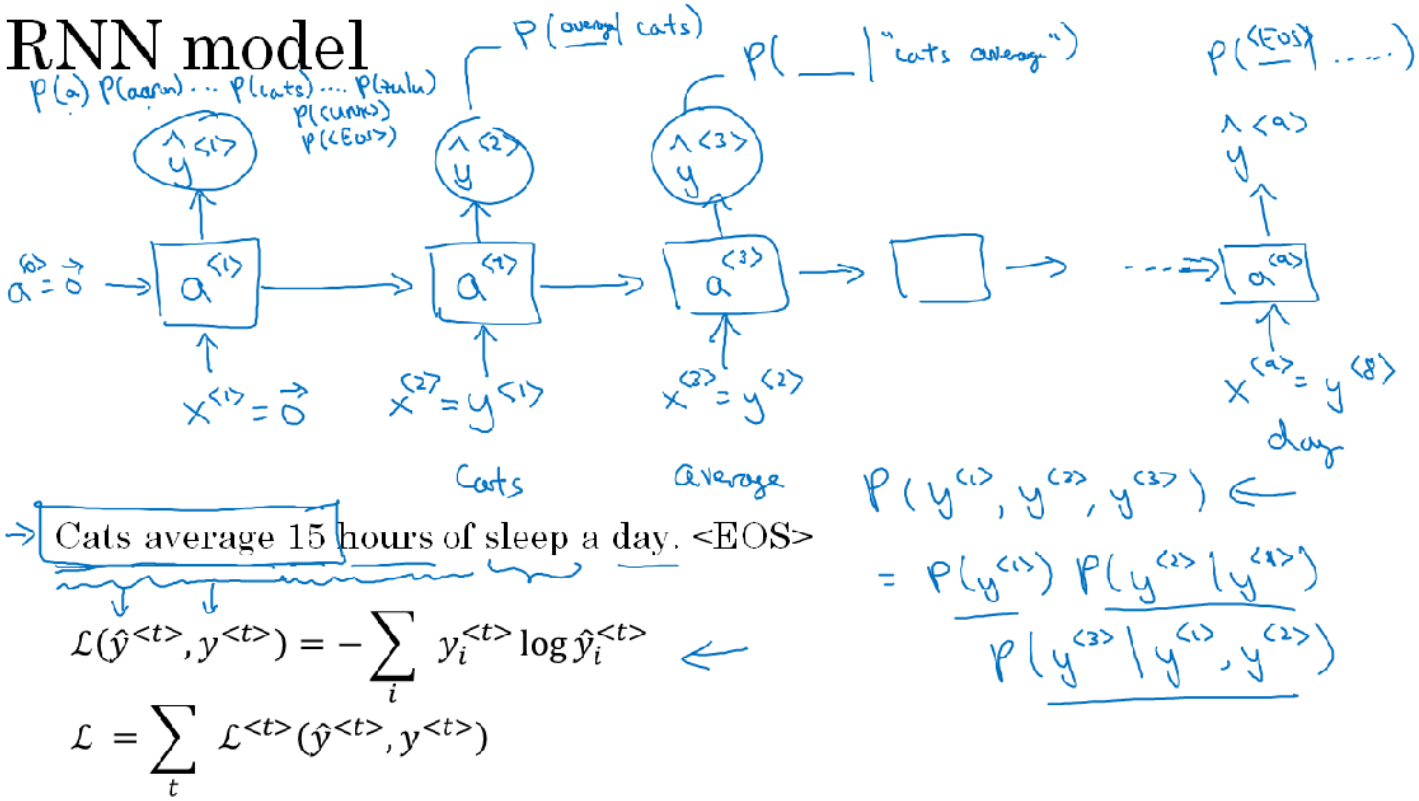
For example, a language model might say that the chance for the first sentence is 3.2×10^{-13} and the chance of the second sentence is say 5.7×10^{-10} and so, with these probabilities, the second sentence is much more likely by over a factor of 10^3 compared to the first sentence and that's why speech recognition system will pick the second choice. So what a language model does is given any sentence is job is to tell you what is the probability of a sentence, of that particular sentence and by probability of sentence I mean, if you want to pick up a random newspaper, open a random email or pick a random webpage or listen to the next thing someone says, the friend of you says. What is the chance that the next sentence you use somewhere out there in the world will be a particular sentence like the apple and pear salad? and this is a fundamental component for both speech recognition systems as you've just seen, as well as for machine translation systems where translation systems wants output only sentences that are likely and so the basic job of a language model is to input a sentence, which I'm going to write as a sequence $y^{<1>}$, $y^{<2>}$ up to $y^{<Ty>}$ and for language model will be useful to represent a sentences as outputs y rather than inputs x. But what the language model does is it estimates the probability of that particular sequence of words.

So how do you build a language model? To build such a model using an RNN you would first need a training set comprising a large corpus of english text. Or text from whatever language you want to build a language model of. And the word corpus is an NLP terminology that just means a large body or a very large set of english text of english sentences. So let's say you get a sentence in your training set as follows.

Language modelling with an RNN



Cats average 15 hours of sleep a day. The first thing you would do is tokenize this sentence and that means you would form a vocabulary as we saw in an earlier section and then map each of these words to, say, one hot vectors, alter indices in your vocabulary. One thing you might also want to do is model when sentences end. So another common thing to do is to add an extra token called a EOS. That stands for End Of Sentence that can help you figure out when a sentence ends. We'll talk more about this later, but the EOS token can be appended to the end of every sentence in your training sets if you want your models explicitly capture when sentences end.



So in this example, we have y1, y2, y3, 4, 5, 6, 7, 8, 9. Nine inputs in this example if you append the end of sentence token to the end. And doing the tokenization step, you can decide whether or not the period should be a token as well. In this example, I'm just ignoring punctuation. So I'm just using

day as another token. And omitting the period, if you want to treat the period or other punctuation as explicit token, then you can add the period to you vocabulary as well. Now, one other detail would be what if some of the words in your training set, are not in your vocabulary. So if your vocabulary uses 10,000 words, maybe the 10,000 most common words in English, then the term Mau as in the Egyptian Mau is a breed of cat, that might not be in one of your top 10,000 tokens. So in that case you could take the word Mau and replace it with a unique token called UNK or stands for unknown words and would just model, the chance of the unknown word instead of the specific word now. Having carried out the tokenization step which basically means taking the input sentence and mapping out to the individual tokens or the individual words in your vocabulary. Next let's build an RNN to model the chance of these different sequences. And one of the things we'll see on the next slide is that you end up setting the inputs x<t> = y<t-1> or you see that in a little bit. So let's go on to built the RNN model and I'm going to continue to use this sentence as the running example. This will be an RNN architecture. At time 0 you're going to end up computing some activation a1 as a function of some inputs x1, and x1 will just be set it to the set of all zeroes, to 0 vector. And the previous A0, by convention, also set that to vector zeroes. But what A1 does is it will make a soft max prediction to try to figure out what is the probability of the first words y. And so that's going to be y<1>. So what this step does is really, it has a soft max it's trying to predict. What is the probability of any word in the dictionary? That the first one is a, what's the chance that the first word is Aaron? And then what's the chance that the first word is cats? All the way to what's the chance the first word is Zulu? Or what's the first chance that the first word is an unknown word? Or what's the first chance that the first word is the in the sentence they'll have, shouldn't have to read? Right, so y hat 1 is output to a soft max, it just predicts what's the chance of the first word being, whatever it ends up being. And in our example, it wind up being the word cats, so this would be a 10,000 way soft max output, if you have a 10,000-word vocabulary. Or 10,002, I guess you could call unknown word and the sentence is two additional tokens. Then, the RNN steps forward to the next step and has some activation, a<1> to the next step. And at this step, his job is try figure out, what is the second word?

But now we will also give it the correct first word. So we'll tell it that, gee, in reality, the first word was actually Cats so that's y1. So tell it cats, and this is why y1 = x2, and so at the second step the output is again predicted by a soft max. The RNN's jobs to predict was the chance of a being whatever the word it is. Is it a or Aaron, or Cats or Zulu or unknown whether EOS or whatever given what had come previously. So in this case, I guess the right answer was average since the sentence starts with cats average. And then you go on to the next step of the RNN. Where you now compute a3. But to predict what is the third word, which is 15, we can now give it the first two words. So we're going to tell it cats average are the first two words. So this next input here, x<3> = y<2>, so the word average is input, and this job is to figure out what is the next word in the sequence. So in other words trying to figure out what is the probability of anywhere than dictionary given that what just came before was cats.

Average, right? And in this case, the right answer is 15 and so on.

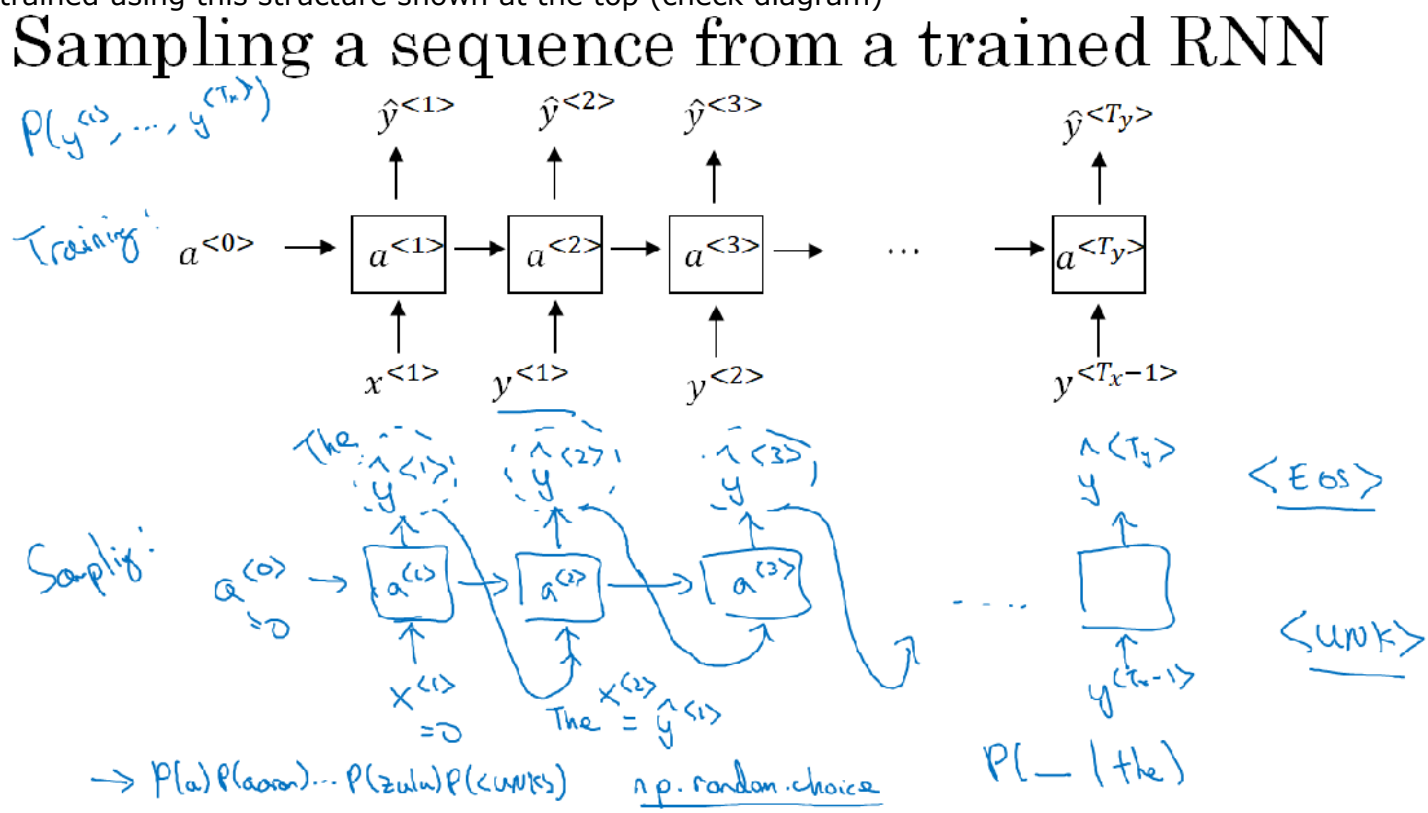
Until at the end, you end up at, I guess, time step 9, you end up feeding it x(9), which is equal to y(8), which is the word, day. And then this has A(9), and its jpob iws to output y hat 9, and this happens to be the EOS token. So what's the chance of whatever this given, everything that comes before, and hopefully it will predict that there's a high chance of it, EOS and the sentence token. So each step in the RNN will look at some set of preceding words such as, given the first three words, what is the distribution over the next word? And so this RNN learns to predict one word at a time going from left to right. Next to train us to a network, we're going to define the cos function. So, at a certain time, t, if the true word was yt and the new networks soft max predicted some y hat t, then this is the soft max loss function that you should already be familiar with. And then the overall loss is just the sum overall time steps of the loss associated with the individual predictions. And if you train this RNN on the last training set, what you'll be able to do is given any initial set of words, such as cats average 15 hours of, it can predict what is the chance of the next word.

And given a new sentence say, y(1), y(2), y(3)with just a three words, for simplicity, the way you can figure out what is the chance of this entire sentence would be. Well, the first soft max tells you what's the chance of y(1). That would be this first output. And then the second one can tell you what's the chance of p of y(2) given y(1). And then the third one tells you what's the chance of y(3) given y(1) and y(2). And so by multiplying out these three probabilities. And you'll see much more details of this

in the previous exercise. By multiply these three, you end up with the probability of the three sentence, of the three-word sentence. So that's the basic structure of how you can train a language model using an RNN.

Sampling novel sequences

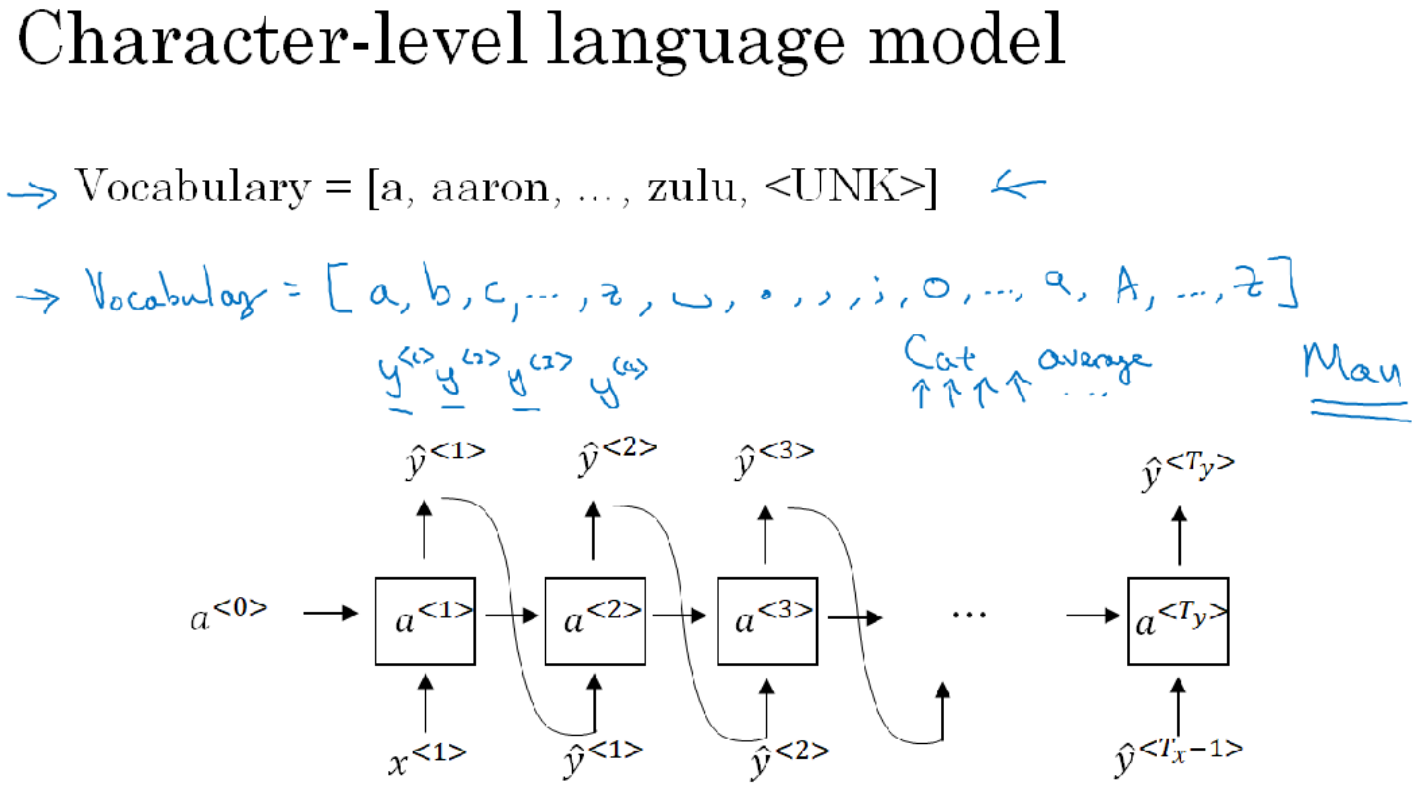
After you train a sequence model, one of the ways you can informally get a sense of what is learned is to have a sample novel sequences. Let's take a look at how you could do that. So remember that a sequence model, models the chance of any particular sequence of words as follows, and so what we like to do is sample from this distribution to generate noble sequences of words. So the network was trained using this structure shown at the top (check diagram)



But to sample, you do something slightly different, so what you want to do is first sample what is the first word you want your model to generate and so for that you input the usual x<1> equals 0, a<0> equals 0. And now your first time stamp will have some max probability over possible outputs. So what you do is you then randomly sample according to this soft max distribution. So what the soft max distribution gives you is it tells you what is the chance that it refers to this word "a", what is the chance that it refers to word "Aaron"? What's the chance it refers to "Zulu", what is the chance that the first word is the Unknown word token. Maybe it was a chance it was a end of sentence token and then you take this vector and use, for example, the numpy command **np.random.choice** to sample according to distribution defined by the vector probabilities, and that lets you sample the first words. Next you then go on to the second time step, and now remember that the second time step is expecting this y<1> as input but what you do is you then take the y1 hat that you just sampled and pass that in here as the input to the next timestep. So whatever works, you just chose the first time step passes this input in the second position, and then this soft max will make a prediction for what is y hat 2.

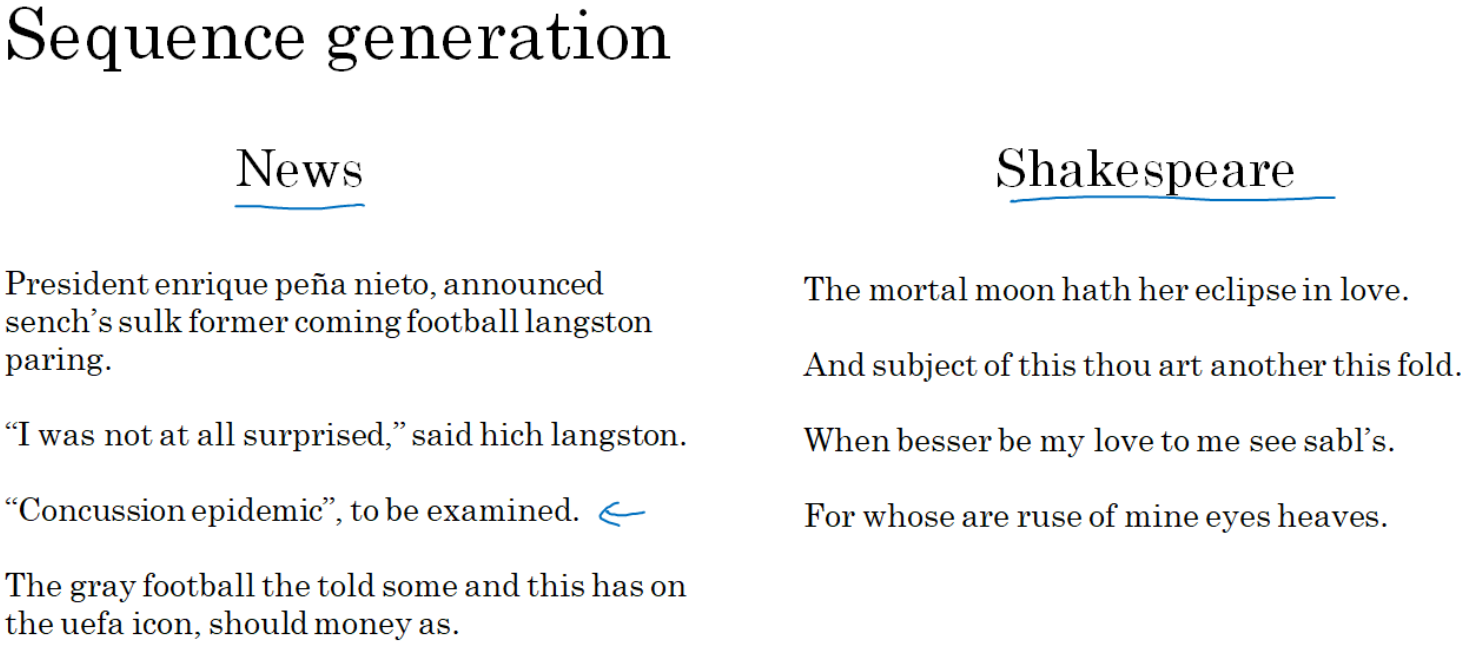
Example, let's say that after you sample the first word, the first word happened to be z, which is very common choice of first word. Then you pass in v as x2, which is now equal to y hat 1 and now you're trying to figure out what is the chance of what the second word is given that the first word is d. And this is going to be y hat 2. Then you again use this type of sampling function to sample y hat 2. And then at the next time stamp, you take whatever choice you had represented say as a one hard encoding. And pass that to next timestep and then you sample the third word to that whatever you chose, and you keep going until you get to the last time step. And so how do you know when the sequence ends? Well, one thing you could do is if the end of sentence token is part of your

vocabulary, you could keep sampling until you generate an EOS token and that tells you you've hit the end of a sentence and you can stop. Or alternatively, if you do not include this in your vocabulary then you can also just decide to sample 20 words or 100 words or something, and then keep going until you've reached that number of time steps and this particular procedure will sometimes generate an unknown word token. If you want to make sure that your algorithm never generates this token, one thing you could do is just reject any sample that came out as unknown word token and just keep resampling from the rest of the vocabulary until you get a word that's not an unknown word. Or you can just leave it in the output as well if you don't mind having an unknown word output. So this is how you would generate a randomly chosen sentence from your RNN language model. Now, so far we've been building a words level RNN, by which I mean the vocabulary are words from English. Depending on your application, one thing you can do is also build a character level RNN.



So in this case your vocabulary will just be the alphabets. Up to z, and as well as maybe space, punctuation if you wish, the digits 0 to 9 and if you want to distinguish the uppercase and lowercase, you can include the uppercase alphabets as well, and one thing you can do as you just look at your training set and look at the characters that appears there and use that to define the vocabulary and if you build a character level language model rather than a word level language model, then your sequence y1, y2, y3, would be the individual characters in your training data, rather than the individual words in your training data. So for our previous example, the sentence cats average 15 hours of sleep a day. In this example, c would be y1, a would be y2, t will be y3, the space will be y4 and so on. Using a character level language model has some pros and cons. **One is that you don't ever have to worry about unknown word tokens. In particular, a character level language model is able to assign a sequence like mau, a non-zero probability. Whereas if mau was not in your vocabulary for the word level language model, you just have to assign it the unknown word token. But the main disadvantage of the character level language model is that you end up with much longer sequences. So many english sentences will have 10 to 20 words but may have many, many dozens of characters. And so character language models are not as good as word level language models at capturing long range dependencies between how the the earlier parts of the sentence also affect the later part of the sentence. And character level models are also just more computationally expensive to train.** So the trend I've been seeing in natural language processing is that for the most part, word level language model are still used, but as computers gets faster there are more and more applications where people are, at least in some special cases, starting to look at more character level models. But they tend to be much hardware, much more computationally expensive to train, so they

are not in widespread use today. Except for maybe specialized applications where you might need to deal with unknown words or other vocabulary words a lot. Or they are also used in more specialized applications where you have a more specialized vocabulary. So under these methods, what you can now do is build an RNN to look at the purpose of English text, build a word level, build a character language model, sample from the language model that you've trained. So here are some examples of text that were examples from a language model, actually from a culture level language model.

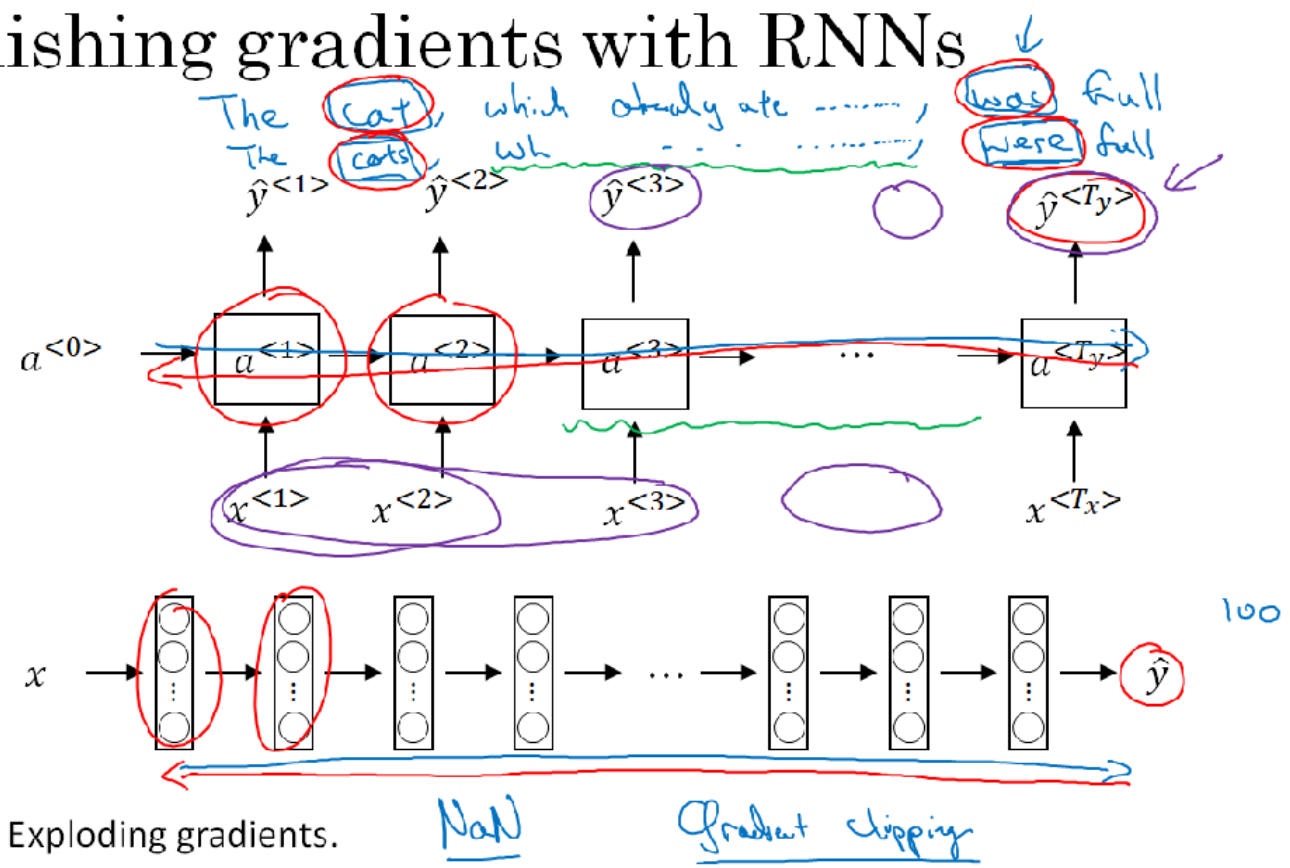


If the model was trained on news articles, then it generates texts like that shown on the left. And this looks vaguely like news text, not quite grammatical, but maybe sounds a little bit like things that could be appearing news, concussion epidemic to be examined. And it was trained on Shakespearean text and then it generates stuff that sounds like Shakespeare could have written it. The mortal moon hath her eclipse in love. And subject of this thou art another this fold. When besser be my love to me see sabl's. For whose are ruse of mine eyes heaves.

So that's it for the basic RNN, and how you can build a language model using it, as well as sample from the language model that you've trained. In the next few sections, we'lll discuss further some of the challenges of training RNNs, as well as how to adjust some of these challenges, specifically vanishing gradients by building even more powerful models of the RNN. So in the next section we'll talk about the problem of vanishing the gradient and we will go on to talk about the GRU, Gate Recurring Unit as well as the LSTM models.

Vanishing gradients with RNNs

We've learned about how RNNs work and how they can be applied to problems like name entity recognition, as well as to language modeling, and we saw how backpropagation can be used to train in RNN. It turns out that one of the problems with a basic RNN algorithm is that it runs into **vanishing gradient problems**. Let's discuss that, and then in the next few sections, we'll talk about some solutions that will help to address this problem. So, you've seen pictures of RNNS that look like diagram shown below:



and let's take a language modeling example. Let's say you see this sentence, "The cat which already ate and maybe already ate a bunch of food that was delicious dot, dot, dot, dot, was full." And so, to be consistent, just because cat is singular, it should be the cat was, were then was, "The cats which already ate a bunch of food was delicious, and apples, and pears, and so on, were full." So to be consistent, it should be cat was or cats were. And this is one example of when language can have very long-term dependencies, where it worked at this much earlier can affect what needs to come much later in the sentence. But it turns out the basics RNN we've seen so far it's not very good at capturing very long-term dependencies. To explain why, you might remember from our early discussions of training very deep neural networks, that we talked about the vanishing gradients problem. So the diagram shown is a very, very deep neural network say, 100 layers or even much deeper than you would carry out forward prop, from left to right and then back prop and we said that, if this is a very deep neural network, then the gradient from just output y, would have a very hard time propagating back to affect the weights of these earlier layers, to affect the computations in the earlier layers and for an RNN with a similar problem, you have forward prop came from left to right, and then back prop, going from right to left and it can be quite difficult, because of the same vanishing gradients problem, for the outputs of the errors associated with the later time steps to affect the computations that are earlier and so in practice, what this means is, it might be difficult to get a neural network to realize that it needs to memorize the just see a singular noun or a plural noun, so that later on in the sequence that can generate either was or were, depending on whether it was singular or plural and notice that in English, this stuff in the middle could be arbitrarily long, right? So you might need to memorize the singular/plural for a very long time before you get to use that bit of information. So because of this problem, the basic RNN model has many local influences, meaning that the output y^<3> is mainly influenced by values close to y^<3> and a value here is mainly influenced by inputs that are somewhere close and it's difficult for the output here to be strongly influenced by an input that was very early in the sequence and this is because whatever the output is, whether this got it right, this got it wrong, it's just very difficult for the area to backpropagate all the way to the beginning of the sequence, and therefore to modify how the neural network is doing computations earlier in the sequence. So this is a weakness of the basic RNN algorithm. One, which was not addressed in the next few sections but if we don't address it, then RNNs tend not to be very good at capturing long-range dependencies and even though this discussion

has focused on vanishing gradients, you will remember when we talked about very deep neural networks, that we also talked about exploding gradients. We're doing back prop, the gradients should not just decrease exponentially, they may also increase exponentially with the number of layers you go through. **It turns out that vanishing gradients tends to be the bigger problem with training RNNs, although when exploding gradients happens, it can be catastrophic because the exponentially large gradients can cause your parameters to become so large that your neural network parameters get really messed up**. So it turns out that exploding gradients are easier to spot because the parameters just blow up and you might often see NaNs, or not a numbers, meaning results of a numerical overflow in your neural network computation and if you do see exploding gradients, one solution to that is apply gradient clipping. and what that really means, all that means is look at your gradient vectors, and if it is bigger than some threshold, re-scale some of your gradient vector so that is not too big. So there are clips according to some maximum value. So if you see exploding gradients, if your derivatives do explode or you see NaNs, just apply gradient clipping, and that's a relatively robust solution that will take care of exploding gradients. But vanishing gradients is much harder to solve and it will be the subject of the next few sections.

So to summarize, in an earlier course, you saw how the training of very deep neural network, you can run into a vanishing gradient or exploding gradient problems with the derivative, either decreases exponentially or grows exponentially as a function of the number of layers and in RNN, say in RNN processing data over a thousand times sets, over 10,000 times sets, that's basically a 1,000 layer or they go 10,000 layer neural network, and so, it too runs into these types of problems. **Exploding gradients, you could sort of address by just using gradient clipping, but vanishing gradients will take more work to address**. So what we do in the next sections is talk about GRU, the greater recurrent units, which is a very effective solution for addressing the vanishing gradient problem and will allow your neural network to capture much longer range dependencies.

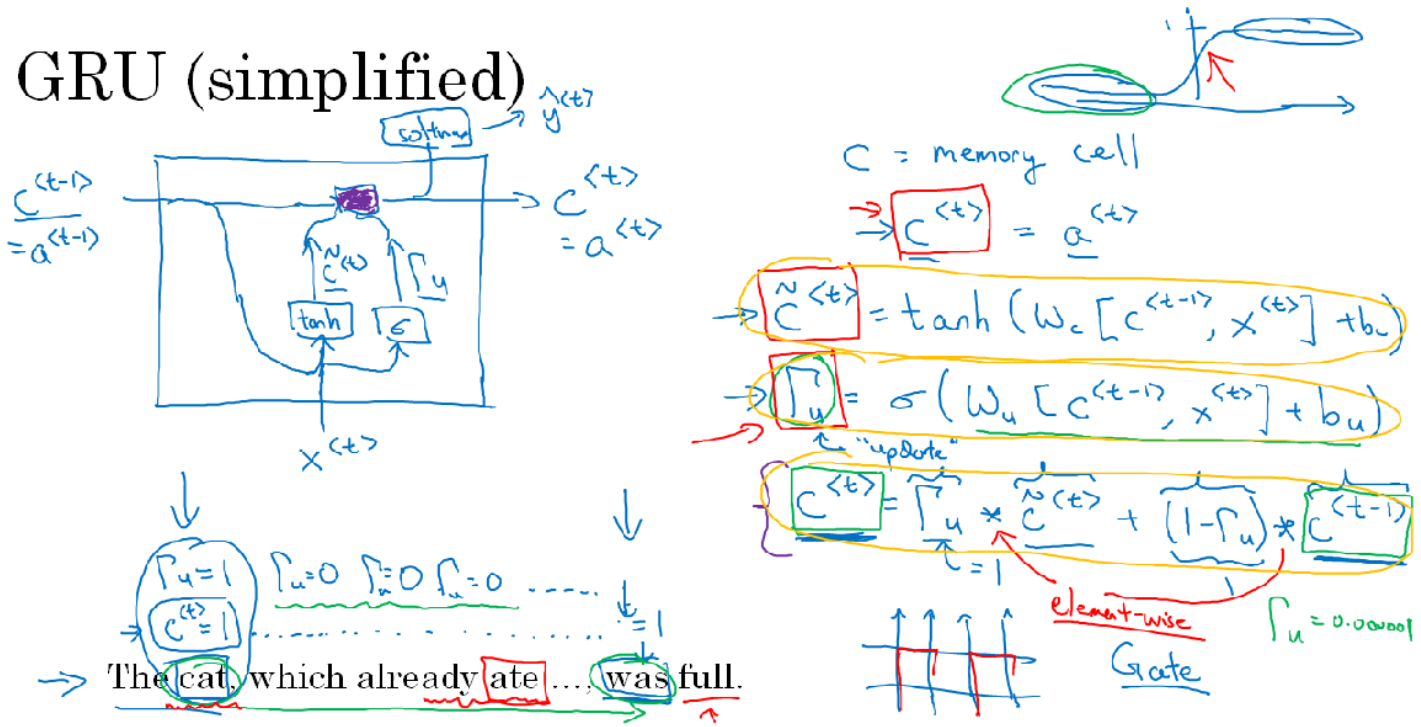
Gated Recurrent Unit (GRU)

You've seen how a basic RNN works. In this section, you learn about the **Gated Recurrent Unit** which is a modification to the RNN hidden layer that makes it much better capturing long range connections and helps a lot with the **vanishing gradient problems**. Let's take a look.

we've already seen the formula for computing the activations at time t of RNN. It's the activation function applied to the parameter Wa times the activations in the previous time set, the current input and then plus ba. So we're going to draw this as a picture. So the RNN unit, we're going to draw as a picture,
drawn as a box which inputs a of t-1, the activation for the last time-step and also inputs x<t> and these two go together and after some weights and after this type of linear calculation, if g is a tanh activation function, then after the tanh, it computes the output activation a. And the output activation a(t) might also be passed to say a softener unit or something that could then be used to output y<t>. So this is maybe a visualization of the RNN unit of the hidden layer of the RNN in terms of a picture. And I want to show you this picture because we're going to use a similar picture to explain the GRU or the Gated Recurrent Unit. Lots of the idea of GRU were due to these two papers respectively by Yu Young Chang, Kagawa, Gaza Hera, Chang Hung Chu and Jose Banjo. And I'm sometimes going to refer to this sentence which we'd seen in the last section to motivate that. Given a sentence like this, you might need to remember the cat was singular, to make sure you understand why that was rather than were. So the cat was for or the cats were for. So as we read in this sentence from left to right, the GRU unit is going to have a new variable called c, which stands for cell, for memory cell. And what the memory cell do is it will provide a bit of memory to remember, for example, whether cat was singular or plural, so that when it gets much further into the sentence it can still work under consideration whether the subject of the sentence was singular or plural. And so at time t the memory cell will have some value c of t. And what we see is that the GRU unit will actually output an activation value a of t that's equal to c of t. And for now I wanted to use different symbol c and a to denote the memory cell value and the output activation value, even though they are the same. I'm using this notation because when we talk about LSTMs, a little bit later, these will be two different values. But for now, for the GRU, c of t is equal to the output activation a of t. So these are the equations that govern the computations of a GRU unit. And every time-step, we're going to consider overwriting the memory cell with a value c tilde of t. So this is going to be a

candidate for replacing c of t. And we're going to compute this using an activation function tanh of Wc. And so that's the parameter to make sure it's Wc and we'll plus this parameter matrix, the previous value of the memory cell, the activation value as well as the current input value x<t>, and then plus the bias. So c tilde of t is going to be a candidate for replacing c<t>. And then the key, really the important idea of the GRU it will be that we have a gate. So the gate, I'm going to call gamma u. This is the capital Greek alphabet gamma subscript u, and u stands for update gate, and this will be a value between zero and one. And to develop your intuition about how GRUs work, think of gamma u, this gate value, as being always zero or one. Although in practice, your compute it with a sigmoid function applied to this. So remember that the sigmoid function looks like this. And so it's value is always between zero and one. And for most of the possible ranges of the input, the sigmoid function is either very, very close to zero or very, very close to one. So for intuition, think of gamma as being either zero or one most of the time. And this alphabet u stands for- I chose the alphabet gamma for this because if you look at a gate fence, looks a bit like this I guess, then there are a lot of gammas in this fence. So that's why gamma u, we're going to use to denote the gate. Also Greek alphabet G, right. G for gate. So G for gamma and G for gate. And then next, the key part of the GRU is this equation which is that we have come up with a candidate where we're thinking of updating c using c tilde, and then the gate will decide whether or not we actually update it. And so the way to think about it is maybe this memory cell c is going to be set to either zero or one depending on whether the word you are considering, really the subject of the sentence is singular or plural. So because it's singular, let's say that we set this to one. And if it was plural, maybe we would set this to zero, and then the GRU unit would memorize the value of the c<t> all the way until here, where this is still equal to one and so that tells it, oh, it's singular so use the choice was. And the job of the gate, of gamma u, is to decide when do you update these values. In particular, when you see the phrase, the cat, you know they you're talking about a new concept the especially subject of the sentence cat. So that would be a good time to update this bit and then maybe when you're done using it, the cat blah blah blah was full, then you know, okay, I don't need to memorize anymore, I can just forget that. So the specific equation we'll use for the GRU is the following. Which is that the actual value of c<t> will be equal to this gate times the candidate value plus one minus the gate times the old value, c<t> minus one. So you notice that if the gate, if this update value, this equal to one, then it's saying set the new value of c<t> equal to this candidate value. So that's like over here, set gate equal to one so go ahead and update that bit. And then for all of these values in the middle, you should have the gate equals zero. So this is saying don't update it, don't update it, don't update it, just hang onto the old value. Because if gamma u is equal to zero, then this would be zero, and this would be one. And so it's just setting c<t> equal to the old value, even as you scan the sentence from left to right. So when the gate is equal to zero, we're saying don't update it, don't update it, just hang on to the value and don't forget what this value was. And so that way even when you get all the way down here, hopefully you've just been setting c<t> equals c<t> minus one all along. And it still memorizes, the cat was singular. So let me also draw a picture to denote the GRU unit. And by the way, when you look in online blog posts and textbooks and tutorials these types of pictures are quite popular for explaining GRUs as well as we'll see later, LSTM units. I personally find the equations easier to understand in a pictures. So if the picture doesn't make sense. Don't worry about it, but I'll just draw in case helps some of you. So a GRU unit inputs c<t> minus one, for the previous time-step and just happens to be equal to 80 minus one. So take that as input and then it also takes as input x<t>, then these two things get combined together. And with some appropriate weighting and some tanh, this gives you c tilde t which is a candidate for placing c<t>, and then with a different set of parameters and through a sigmoid activation function, this gives you gamma u, which is the update gate. And then finally, all of these things combine together through another operation. And I won't write out the formula, but this box here which wish I shaded in purple represents this equation which we had down there. So that's what this purple operation represents. And it takes as input the gate value, the candidate new value, or there is this gate value again and the old value for c<t>, right. So it takes as input this, this and this and together they generate the new value for the memory cell. And so that's c<t> equals a. And if you wish you could also use this process to soft max or something to make some prediction for y<t>. So that is the GRU unit or at least a slightly simplified version of it. And what is remarkably good at is through the gates deciding that when you're scanning the sentence from left to right say, that's a good time to update one particular memory cell and then to not change, not change it until you get to the point where you really need it to use this memory cell

that is set even earlier in the sentence. And because the sigmoid value, now, because the gate is quite easy to set to zero right. So long as this quantity is a large negative value, then up to numerical around off the uptake gate will be essentially zero. Very, very, very close to zero. So when that's the case, then this updated equation and subsetting c<t> equals c<t> minus one. And so this is very good at maintaining the value for the cell. And because gamma can be so close to zero, can be 0.000001 or even smaller than that, it doesn't suffer from much of a vanishing gradient problem. Because when you say gamma so close to zero this becomes essentially c<t> equals c<t> minus one and the value of c<t> is maintained pretty much exactly even across many many many many time-steps. So this can help significantly with the vanishing gradient problem and therefore allow a neural network to go on even very long range dependencies, such as a cat and was related even if they're separated by a lot of words in the middle. Now I just want to talk over some more details of how you implement this. In the equations I've written, c<t> can be a vector. So if you have 100 dimensional or hidden activation value then c<t> can be a 100 dimensional say. And so c tilde t would also be the same dimension, and gamma would also be the same dimension as the other things on drawing boxes. And in that case, these asterisks are actually element wise multiplication. So here if gamma u, if the gate is 100 dimensional vector, what it is really a 100 dimensional vector of bits, the value is mostly zero and one. That tells you of this 100 dimensional memory cell which are the bits you want to update. And, of course, in practice gamma won't be exactly zero or one. Sometimes it takes values in the middle as well but it is convenient for intuition to think of it as mostly taking on values that are exactly zero, pretty much exactly zero or pretty much exactly one. And what these element wise multiplications do is it just element wise tells the GRU unit which other bits in your- It just tells your GRU which are the dimensions of your memory cell vector to update at every time-step. So you can choose to keep some bits constant while updating other bits. So, for example, maybe you use one bit to remember the singular or plural cat and maybe use some other bits to realize that you're talking about food. And so because you're talk about eating and talk about food, then you'd expect to talk about whether the cat is four letter, right. You can use different bits and change only a subset of the bits every point in time. You now understand the most important ideas of the GRU. What I'm presenting in this slide is actually a slightly simplified GRU unit. Let me describe the full GRU unit. So to do that, let me copy the three main equations. This one, this one and this one to the next slide. So here they are. And for the full GRU unit, I'm sure to make one change to this which is, for the first equation which was calculating the candidate new value for the memory cell, I'm going just to add one term. Let me pushed that a little bit to the right, and I'm going to add one more gate. So this is another gate gamma r. You can think of r as standing for relevance. So this gate gamma r tells you how relevant is c<t> minus one to computing the next candidate for c<t>. And this gate gamma r is computed pretty much as you'd expect with a new parameter matrix Wr, and then the same things as input x<t> plus br. So as you can imagine there are multiple ways to design these types of neural networks. And why do we have gamma r? Why not use a simpler version from the previous slides? So it turns out that over many years researchers have experimented with many, many different possible versions of how to design these units, to try to have longer range connections, to try to have more the longer range effects and also address vanishing gradient problems. And the GRU is one of the most commonly used versions that researchers have converged to and found as robust and useful for many different problems. If you wish you could try to invent new versions of these units if you want, but the GRU is a standard one, that's just common used. Although you can imagine that researchers have tried other versions that are similar but not exactly the same as what I'm writing down here as well. And the other common version is called an LSTM which stands for Long Short Term Memory which we'll talk about in the next video. But GRUs and LSTMs are two specific instantiations of this set of ideas that are most commonly used. Just one note on notation. I tried to define a consistent notation to make these ideas easier to understand. If you look at the academic literature, you sometimes see people- If you look at the academic literature sometimes you see people using alternative notation to be x tilde, u, r and h to refer to these quantities as well. But I try to use a more consistent notation between GRUs and LSTMs as well as using a more consistent notation gamma to refer to the gates, so hopefully make these ideas easier to understand. So that's it for the GRU, for the Gate Recurrent Unit. This is one of the ideas in RNN that has enabled them to become much better at capturing very long range dependencies has made RNN much more effective. Next, as I briefly mentioned, the other most commonly used variation of this class of idea is something called the LSTM unit, Long Short Term Memory unit. Let's take a look at that in the next video.



[Cho et al., 2014. On the properties of neural machine translation: Encoder-decoder approaches]
[Chung et al., 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling]

Full GRU

$$\tilde{c}^{<t>} = \tanh(W_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c)$$

$$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$$

$$\Gamma_r = \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r)$$

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) + c^{<t-1>}$$

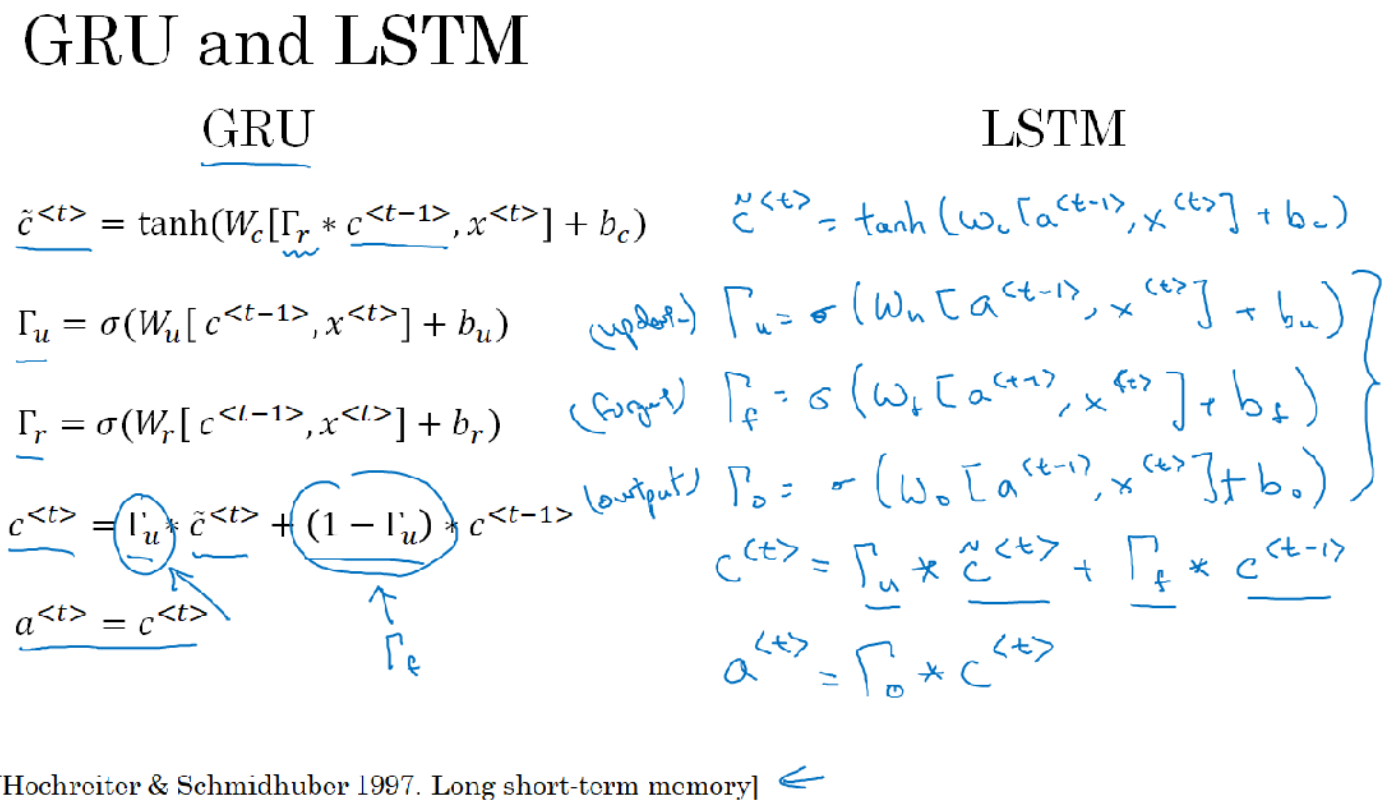
LSTM

The cat, which ate already, was full.

Long Short Term Memory (LSTM)

In the last section, we've learned about the GRU, the **gated recurrent units**, and how that can allow you to learn very long range connections in a sequence. The other type of unit that allows you to do this very well is the LSTM or the **long short term memory units**, and this is even more

powerful than the GRU. Let's take a look. Below are the equations from the previous section for the GRU.



[Hochreiter & Schmidhuber 1997. Long short-term memory]

And for the GRU, we had $a^{<t>}$ equals $c^{<t>}$, and two gates, the optic gate and the relevance gate, $\tilde{c}^{<t>}$, which is a candidate for replacing the memory cell, and then we use the update gate, gamma_u, to decide whether or not to update $c^{<t>}$using $\tilde{c}^{<t>}$. The LSTM is an even slightly more powerful and more general version of the GRU, and is due to Sepp Hochreiter and Jurgen Schmidhuber. And this was a really seminal paper, a huge impact on sequence modelling. I think this paper is one of the more difficult to read. It goes quite along into theory of vanishing gradients. And so, I think more people have learned about the details of LSTM through maybe other places than from this particular paper even though I think this paper has had a wonderful impact on the Deep Learning community. But these are the equations that govern the LSTM. So, the book continued to the memory cell, c, and the candidate value for updating it, c_tilde_t, will be this, and so on. Notice that for the LSTM, we will no longer have the case that a_t is equal to c_t. So, this is what we use. And so, this is just like the equation on the left except that with now, more specially use a_t there or a_t minus one instead of c_t minus one. And we're not using this gamma or this relevance gate. Although you could have a variation of the LSTM where you put that back in, but with the more common version of the LSTM, doesn't bother with that. And then we will have an update gate, same as before. So, W updates and we're going to use a_t minus one here, x_t plus b_u. And one new property of the LSTM is, instead of having one update gate control, both of these terms, we're going to have two separate terms. So instead of gamma_u and one minus gamma_u, we're going have gamma_u here. And forget gate, which we're going to call gamma_f. So, this gate, gamma_f, is going to be sigmoid of pretty much what you'd expect, x_t plus b_f. And then, we're going to have a new output gate which is sigma of W_o. And then again, pretty much what you'd expect, plus b_o. And then, the update value to the memory so will be c_t equals gamma u. And this asterisk denotes element-wise multiplication. This is a vector-vector element-wise multiplication, plus, and instead of one minus gamma u, we're going to have a separate forget gate, gamma_f, times c of t minus one. So this gives the memory cell the option of keeping the old value c_t minus one and then just adding to it, this new value, c tilde of t. So, use a separate update and forget gates. So, this stands for update, forget, and output gate. And then finally, instead of a_t equals c_t a_t is a_t equal to the output gate element-wise multiplied by c_t. So, these are the equations that govern the LSTM and you can tell it has three gates instead of two. So, it's a bit more complicated and it places the gates into slightly different places. So, here again are the equations governing the behavior of the LSTM. Once again, it's traditional to explain

these things using pictures. So let me draw one here. And if these pictures are too complicated, don't worry about it. I personally find the equations easier to understand than the picture. But I'll just show the picture here for the intuitions it conveys. The bigger picture here was very much inspired by a blog post due to Chris Ola, title, Understanding LSTM Network, and the diagram drawing here is quite similar to one that he drew in his blog post. But the key thing is to take away from this picture are maybe that you use a_t minus one and x_t to compute all the gate values.

LSTM units

GRU

$$\tilde{c}^{<t>} = \tanh(W_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c)$$

$$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$$

$$\Gamma_r = \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r)$$

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>}$$

$$a^{<t>} = c^{<t>}$$

LSTM

$$\tilde{c}^{<t>} = \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c)$$

$$\Gamma_u = \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u)$$

$$\Gamma_f = \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f)$$

$$\Gamma_o = \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o)$$

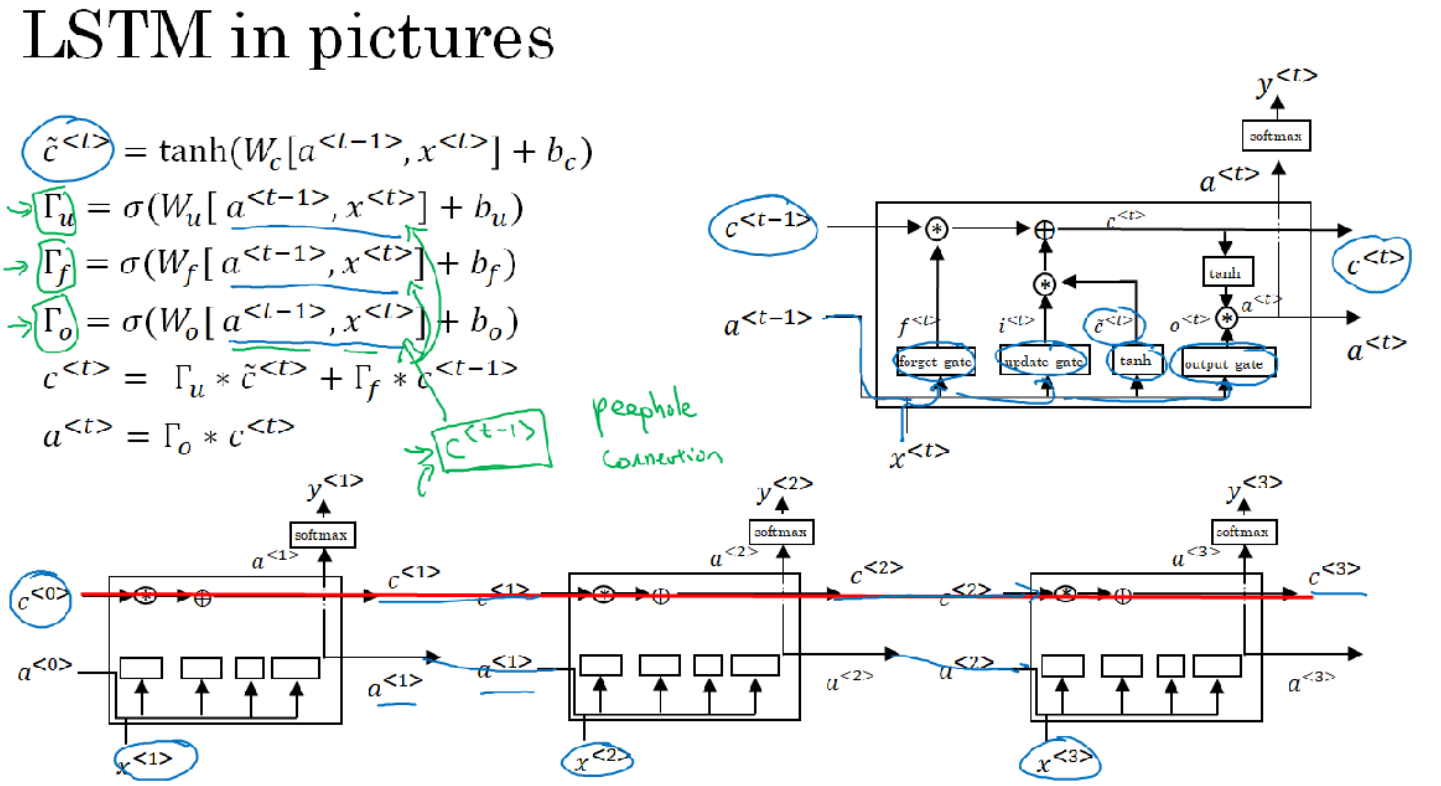
$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>}$$

$$a^{<t>} = \Gamma_o * c^{<t>}$$

[Hochreiter & Schmidhuber 1997. Long short-term memory]

In this picture, you have a_t minus one, x_t coming together to compute the forget gate, to compute the update gates, and to compute output gate. And they also go through a tanh to compute c_tilde_of_t. And then these values are combined in these complicated ways with element-wise multiplies and so on, to get c_t from the previous c_t minus one. Now, one element of this is interesting as you have a bunch of these in parallel. So, that's one of them and you connect them. You then connect these temporally. So it does the input x_1 then x_2, x_3. So, you can take these units and just hold them up as follows, where the output a at the previous timestep is the input a at the next timestep, the c. I've simplified to diagrams a little bit in the bottom. And one cool thing about this you'll notice is that there's this line at the top that shows how, so long as you set the forget and the update gate appropriately, it is relatively easy for the LSTM to have some value c_0 and have that be passed all the way to the right to have your, maybe, c_3 equals c_0. And this is why the LSTM, as well as the GRU, is very good at memorizing certain values even for a long time, for certain real values stored in the memory cell even for many, many timesteps. So, that's it for the LSTM. As you can imagine, there are also a few variations on this that people use. Perhaps, the most common one is that instead of just having the gate values be dependent only on a_t minus one, x_t, sometimes, people also sneak in there the values c_t minus one as well. This is called a peephole connection. Not a great name maybe but you'll see, peephole connection. What that means is that the gate values may depend not just on a_t minus one and on x_t, but also on the previous memory cell value, and the peephole connection can go into all three of these gates' computations. So that's one common variation you see of LSTMs. One technical detail is that these are, say, 100-dimensional vectors. So if you have a 100-dimensional hidden memory cell unit, and so is this. And the, say, fifth element of c_t minus one affects only the fifth element of the corresponding gates, so that relationship is one-to-one, where not every element of the 100-dimensional c_t minus one can affect all elements of the case. But instead, the first element of c_t minus one affects the first element of the case, second element affects the second element, and so on. But if you ever read the paper and see someone talk about the peephole connection, that's when they mean that c_t minus one is used

to affect the gate value as well. So, that's it for the LSTM. When should you use a GRU? And when should you use an LSTM? There isn't widespread consensus in this. And even though I presented GRUs first, in the history of deep learning, LSTMs actually came much earlier, and then GRUs were relatively recent invention that were maybe derived as Pavia's simplification of the more complicated LSTM model. Researchers have tried both of these models on many different problems, and on different problems, different algorithms will win out. So, there isn't a universally-superior algorithm which is why I want to show you both of them. But I feel like when I am using these, the advantage of the GRU is that it's a simpler model and so it is actually easier to build a much bigger network, it only has two gates, so computationally, it runs a bit faster. So, it scales the building somewhat bigger models but the LSTM is more powerful and more effective since it has three gates instead of two. If you want to pick one to use, I think LSTM has been the historically more proven choice. So, if you had to pick one, I think most people today will still use the LSTM as the default first thing to try. Although, I think in the last few years, GRUs had been gaining a lot of momentum and I feel like more and more teams are also using GRUs because they're a bit simpler but often work just as well. It might be easier to scale them to even bigger problems. So, that's it for LSTMs. Well, either GRUs or LSTMs, you'll be able to build neural network that can capture a much longer range depends.



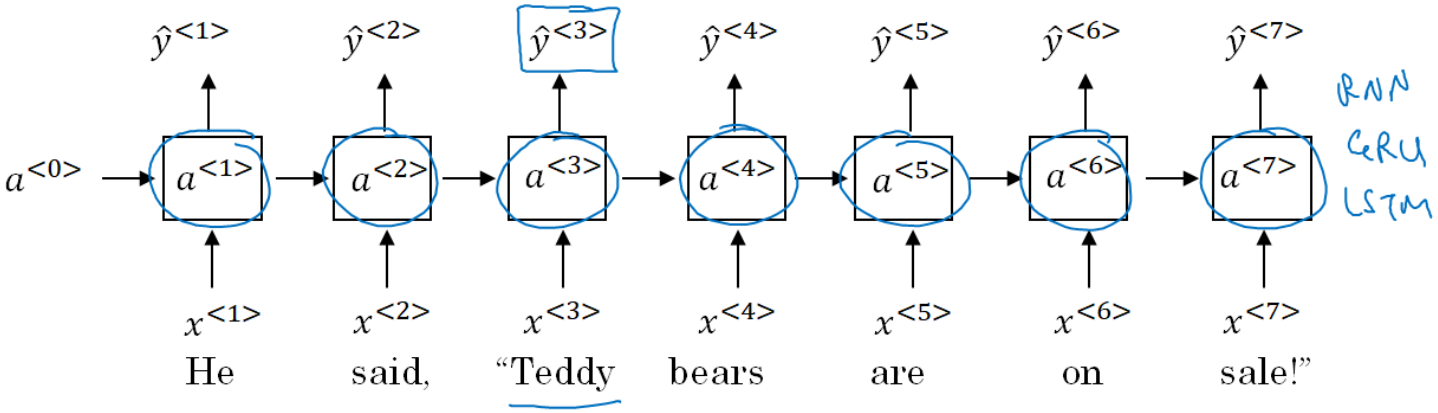
Bidirectional RNN

By now, we've seen most of the key building blocks of RNNs. But, there are just two more ideas that let you build much more powerful models. One is **bidirectional RNNs**, which lets you at a point in time to take information from both earlier and later in the sequence, so we'll talk about that in this section and second, is **deep RNNs**, which we'll see in the next section. So let's start with Bidirectional RNNs. So, to motivate bidirectional RNNs, let's look at this network which you've seen a few times before in the context of named entity recognition and one of the problems of this network is that, to figure out whether the third word Teddy is a part of the person's name, it's not enough to just look at the first part of the sentence.

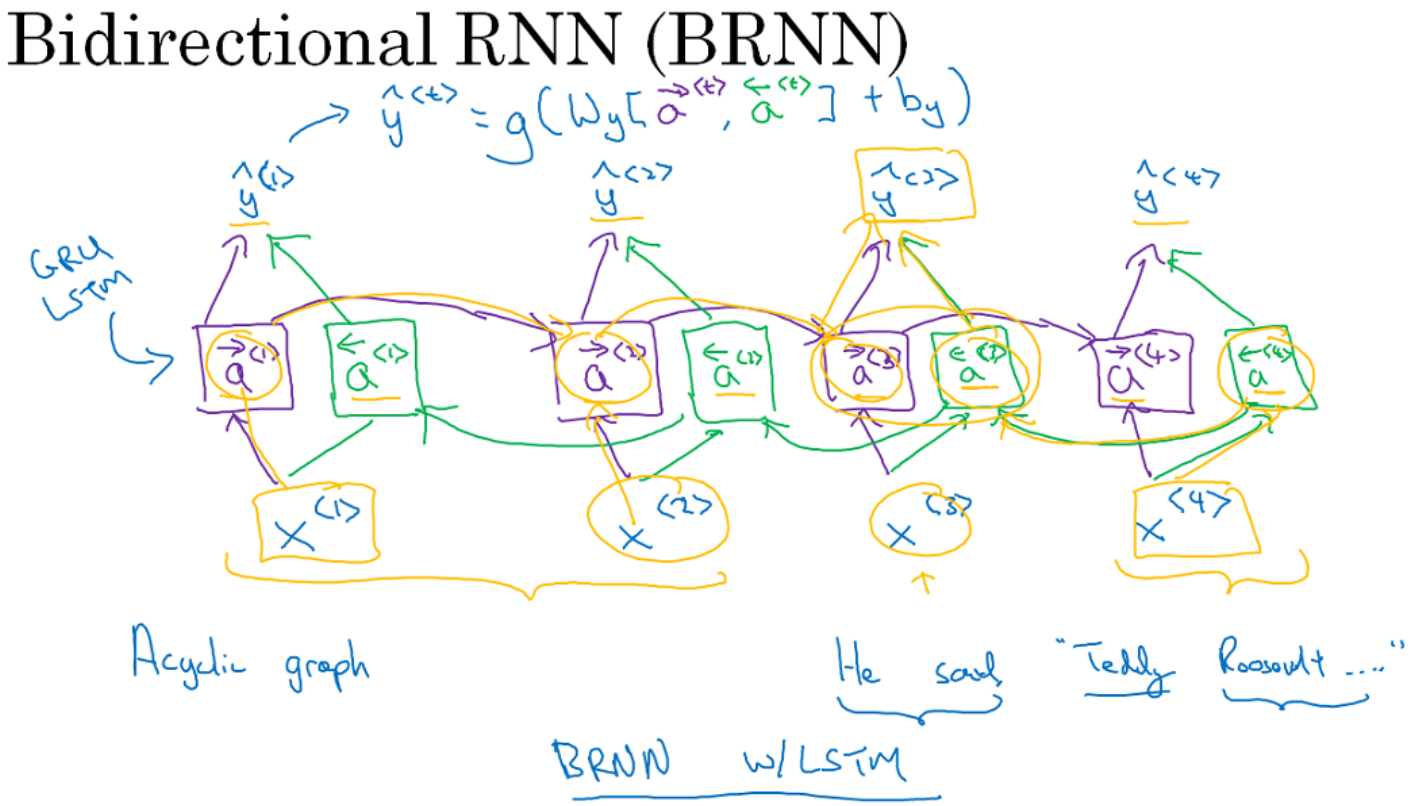
Getting information from the future

He said, "Teddy bears are on sale!"

He said, "Teddy Roosevelt was a great President!"



So to tell, if yhat$^{<3>}$ should be zero or one, you need more information than just the first three words because the first three words doesn't tell you if they'll talking about Teddy bears or talk about the former US president, Teddy Roosevelt. So this is a unidirectional or forward directional only RNN and, this comment I just made is true, whether these cells are standard RNN blocks or whether they're GRU units or whether they're LSTM blocks. But all of these blocks are in a forward only direction. So what a bidirectional RNN does or BRNN, is fix this issue. So, a bidirectional RNN works as follows. I'm going to use a simplified four inputs or maybe a four word sentence. So we have four inputs. x$^{<1>}$ through x$^{<4>}$.



So this networks heading there will have a forward recurrent components. So we're going to call this, a$^{<1>}$, a$^{<2>}$, a$^{<3>}$ and a$^{<4>}$ and we're going to draw a right arrow over that to denote this is the forward recurrent component, and so they'll be connected as shown in above diagram and so, each of

these four recurrent units inputs the current x, and then feeds in to help predict $\hat{y}^{<1>}$, $\hat{y}^{<2>}$, $\hat{y}^{<3>}$, and $\hat{y}^{<4>}$. So far we've drawn the RNN from the previous section, but with the arrows placed in slightly funny positions. But as we draw the arrows in this slightly funny positions because what we're going to do is add a backward recurrent layer. So we'd have $a^{<1>}$, left arrow to denote this is a backward connection, and then $a^{<2>}$ backwards, $a^{<3>}$ backwards and $a^{<4>}$ backwards, so the left arrow denotes that it is a backward connection and so, we're then going to connect to network up as follows and A backward connections will be connected to each other going backward in time. So, notice that this network defines a **Acyclic graph** and so, given an input sequence, $x^{<1>}$ through $x^{<4>}$, the fourth sequence will first compute A forward one, then use that to compute A forward two, then A forward three, then A forward four. Whereas, the backward sequence would start by computing A backward four, and then go back and compute A backward three, and then as you are computing network activation, this is not backward this is forward prop. But the forward prop has part of the computation going from left to right and part of computation going from right to left in this diagram. But having computed A backward three, you can then use those activations to compute A backward two, and then A backward one, and then finally having computed all you had in the activations, you can then make your predictions.

So, for example, to make the predictions, your network will have something like yhat at time t is an activation function applied to WY with both the forward activation at time t, and the backward activation at time t being fed in to make that prediction at time t. So, if you look at the prediction at time set three for example, then information from $x^{<1>}$ can flow through here, forward one to forward two, they're are all stated in the function here, to forward three to $\hat{y}^{<3>}$. So information from $x^{<1>}$, $x^{<2>}$, $x^{<3>}$ are all taken into account with information from $x^{<4>}$ can flow through a backward four to a backward three to $y^{<3>}$. So this allows the prediction at time three to take as input both information from the past, as well as information from the present which goes into both the forward and the backward things at this step, as well as information from the future.

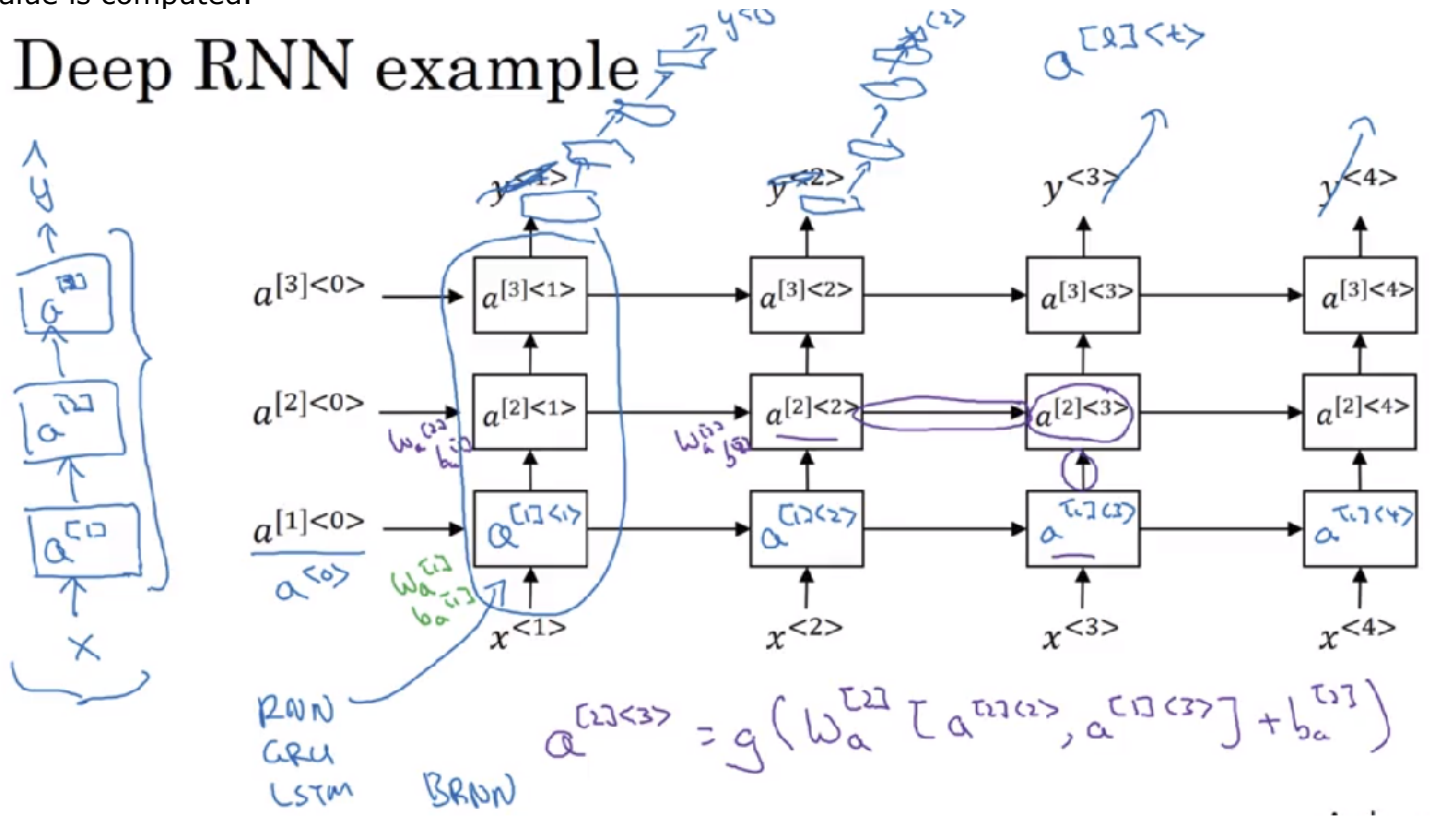
So, in particular, given a phrase like, "He said, Teddy Roosevelt..." To predict whether Teddy is a part of the person's name, you take into account information from the past and from the future. So this is the bidirectional recurrent neural network and these blocks here can be not just the standard RNN block but they can also be GRU blocks or LSTM blocks. In fact, for a lots of NLP problems, for a lot of text with natural language processing problems, **a bidirectional RNN with a LSTM appears to be commonly used. So, we have NLP problem and you have the complete sentence, you try to label things in the sentence, a bidirectional RNN with LSTM blocks both forward and backward would be a pretty views of first thing to try.**

So, that's it for the bidirectional RNN and this is a modification they can make to the basic RNN architecture or the GRU or the LSTM, and by making this change you can have a model that uses RNN and or GRU or LSTM and is able to make predictions anywhere even in the middle of a sequence by taking into account information potentially from the entire sequence. **The disadvantage of the bidirectional RNN is that you do need the entire sequence of data before you can make predictions anywhere.** So, for example, if you're building a speech recognition system, then the BRNN will let you take into account the entire speech utterance but if you use this straightforward implementation, you need to wait for the person to stop talking to get the entire utterance before you can actually process it and make a speech recognition prediction. So for a real type speech recognition applications, they're somewhat more complex modules as well rather than just using the standard bidirectional RNN as you've seen here. **But for a lot of natural language processing applications where you can get the entire sentence all the same time, the standard BRNN algorithm is actually very effective**. So, that's it for BRNNs and next and final SECTION for this week, let's talk about how to take all of these ideas RNNs, LSTMs and GRUs and the bidirectional versions and construct deep versions of them.

Deep RNNs

The different versions of RNNs you've seen so far will already work quite well by themselves. But for learning very complex functions sometimes is useful to stack multiple layers of RNNs together to build even deeper versions of these models. In this section, we'll see how to build these deeper RNNs. Let's take a look. So we remember for a standard neural network, you will have an input x and then that's stacked to some hidden layer and so that might have activations of say, a1 for the first

hidden layer, and then that's stacked to the next layer with activations a2, then maybe another layer, activations a3 and then you make a prediction ŷ. So a deep RNN is a bit like this, by taking this network that (shown in diagram) and unrolling that in time. So let's take a look. So here's the standard RNN that you've seen so far. But we've changed the notation a little bit which is that, instead of writing this as a0 for the activation time zero, I've added this square bracket 1 to denote that this is for layer one. So the notation we're going to use is a[l] to denote that it's an activation associated with layer l and then <t> to denote that that's associated over time t. So this will have an activation on $a^{[1]<1>}$, $a^{[1]<2>}$, $a^{[1]<3>}$, $a^{[1]<4>}$ and then we can just stack these things on top and so this will be a new network with three hidden layers. So let's look at an example of how this value is computed.



So $a^{[2]<3>}$ has two inputs. It has the input coming from the bottom, and there's the input coming from the left. So the computer has an activation function g applied to a way matrix. Check the equation at the end of diagram which calculates the activation value and so the same parameters $Wa^{[2]}$ and $ba^{[2]}$ are used for every one of these computations at this layer. Whereas, in contrast, the first layer would have its own parameters $Wa^{[1]}$ and $ba^{[1]}$. So whereas for standard RNNs like the one on the left, you know we've seen neural networks that are very, very deep, maybe over 100 layers. For RNNs, having three layers is already quite a lot. Because of the temporal dimension, these networks can already get quite big even if you have just a small handful of layers and you don't usually see these stacked up to be like 100 layers. One thing you do see sometimes is that you have recurrent layers that are stacked on top of each other. But then you might take the output here, let's get rid of this, and then just have a bunch of deep layers that are not connected horizontally but have a deep network here that then finally predicts $y^{<1>}$ and you can have the same deep network here that predicts $y^{<2>}$. So this is a type of network architecture that we're seeing a little bit more where you have three recurrent units that connected in time, followed by a network, followed by a network after that, as we seen for $y^{<3>}$ and $y^{<4>}$, of course. There's a deep network, but that does not have the horizontal connections. So that's one type of architecture we seem to be seeing more of and quite often, these blocks don't just have to be standard RNN, the simple RNN model. They can also be GRU blocks LSTM blocks and finally, you can also build deep versions of the bidirectional RNN. Because deep RNNs are quite computationally expensive to train, there's often a large temporal extent already, though you just don't see as many deep recurrent layers. In this case three deep recurrent layers that are connected in time. You don't see as many deep recurrent layers as you would see in a number of layers in a deep conventional neural network. So that's it for deep RNNs. With what you've

seen this week, ranging from the basic RNN, the basic recurrent unit, to the GRU, to the LSTM, to the bidirectional RNN, to the deep versions of this that you just saw, you now have a very rich toolbox for constructing very powerful models for learning sequence models.

Week 2: Natural Language Processing & Word Embeddings

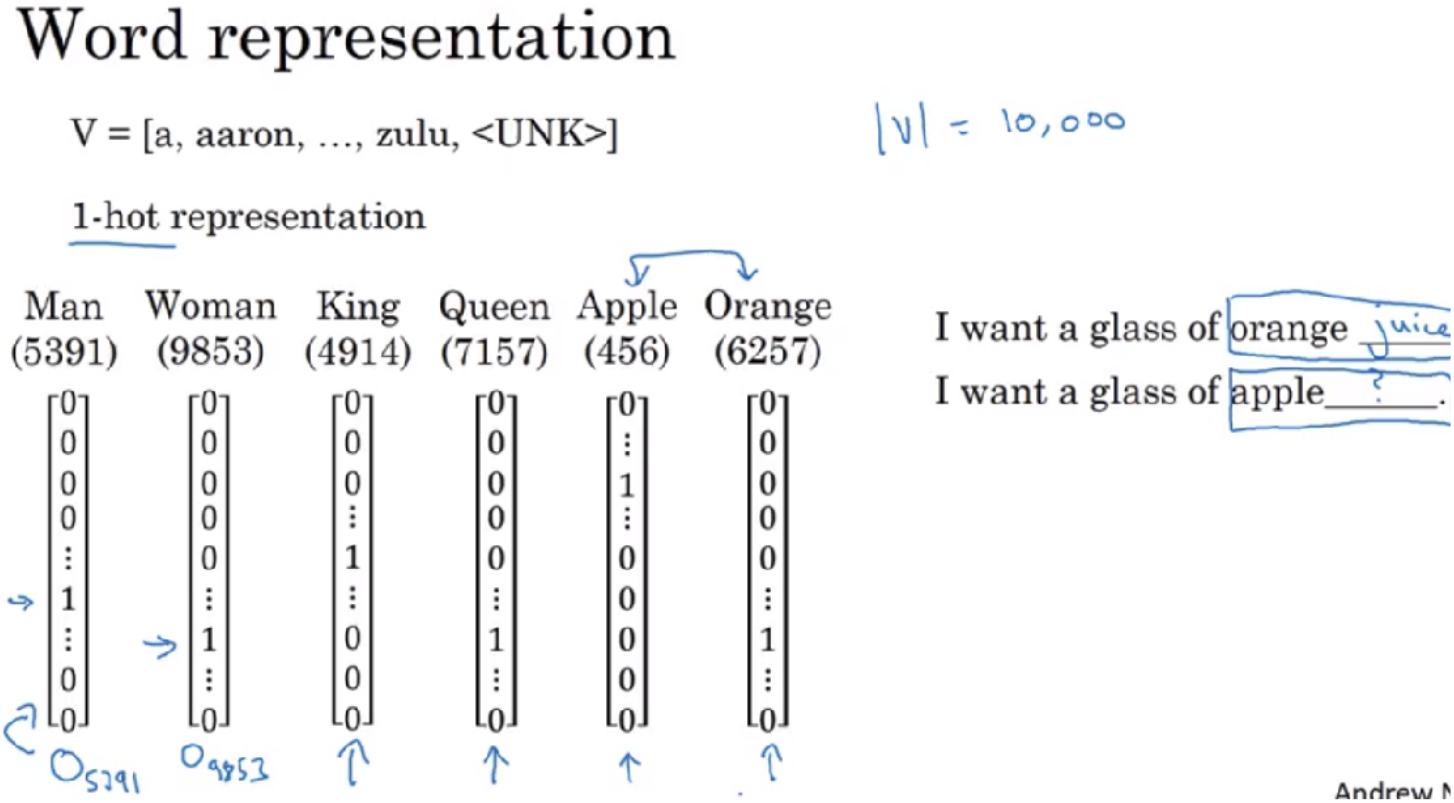
Natural language processing with deep learning is an important combination. Using word vector representations and embedding layers you can train recurrent neural networks with outstanding performances in a wide variety of industries. Examples of applications are sentiment analysis, named entity recognition and machine translation.

Introduction to Word Embeddings

Word Representation

In the last week, we learned about RNNs, GRUs, and LSTMs. In this week, we'll discuss how many of these ideas can be applied to NLP or Natural Language Processing, which is one of the features of AI because it's really being revolutionized by deep learning. One of the key ideas you learn about is word embeddings, which is a way of representing words. The less your algorithms automatically understand analogies like that, man is to woman, as king is to queen, and many other examples and through these ideas of word embeddings, we'll be able to build NLP applications, even can model with usually of relatively small label training sets. Finally towards the end of the week, we'll see how to debias word embeddings. That's to reduce undesirable gender or ethnicity or other types of bias that learning algorithms can sometimes pick up. So with that, let's get started with a discussion on word representation.

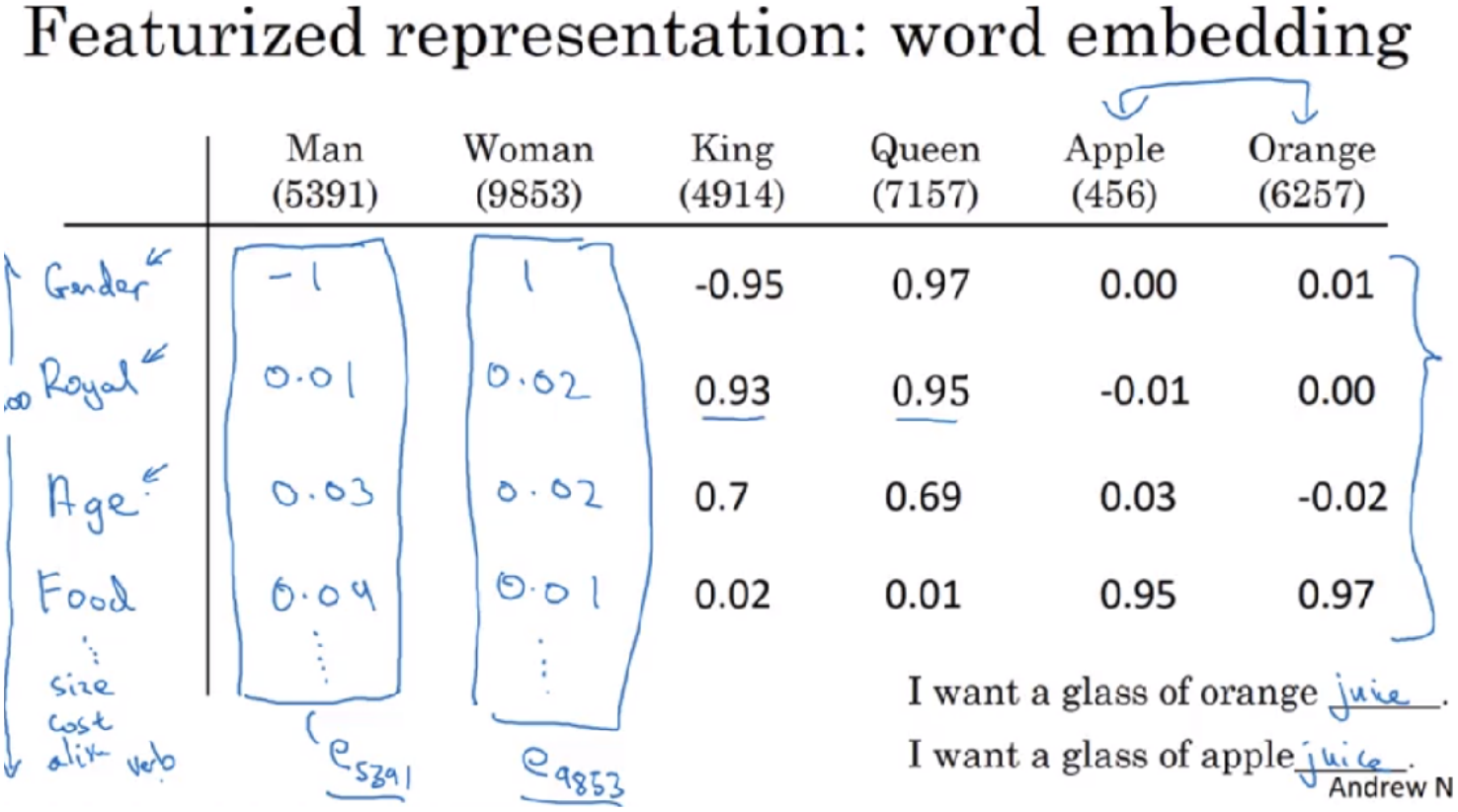
So far, we've been representing words using a vocabulary of words, and a vocabulary from the previous week might be say, 10,000 words. And we've been representing words using a one-hot vector. So for example,



if man is word number 5391 in this dictionary, then you represent him with a vector with one in position 5391. And I'm also going to use O subscript 5391 to represent this factor, where O here stands for one-hot. And then, if woman is word number 9853, then you represent it with O subscript 9853 which just has a one in position 9853 and zeros elsewhere. And then other words king, queen, apple, orange will be similarly represented with one-hot vector. One of the weaknesses of this representation is that it treats each word as a thing unto itself, and it doesn't allow an algorithm to easily generalize the cross words. For example, let's say you have a language model that has learned

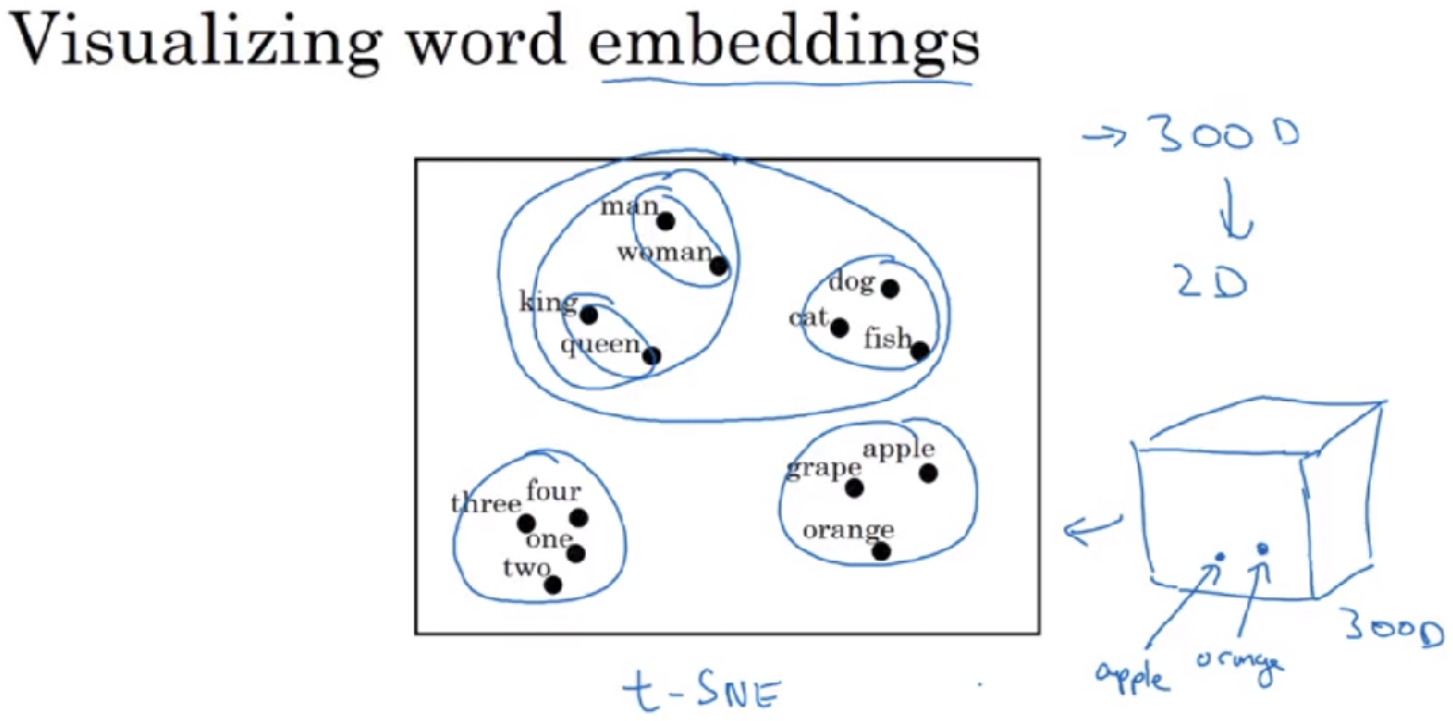
that when you see I want a glass of orange blank. Well, what do you think the next word will be? Very likely, it'll be juice. But even if the learning algorithm has learned that I want a glass of orange juice is a likely sentence, if it sees I want a glass of apple blank. As far as it knows the relationship between apple and orange is not any closer as the relationship between any of the other words man, woman, king, queen, and orange. And so, it's not easy for the learning algorithm to generalize from knowing that orange juice is a popular thing, to recognizing that apple juice might also be a popular thing or a popular phrase. And this is because the any product between any two different one-hot vector is zero. If you take any two vectors say, queen and king and any product of them, the end product is zero. If you take apple and orange and any product of them, the end product is zero. And you couldn't distance between any pair of these vectors is also the same. So it just doesn't know that somehow apple and orange are much more similar than king and orange or queen and orange.

So, won't it be nice if instead of a one-hot presentation we can instead learn a featured representation with each of these words, a man, woman, king, queen, apple, orange or really for every word in the dictionary, we could learn a set of features and values for each of them.



So for example, each of these words, we want to know what is the gender associated with each of these things. So, if gender goes from minus one for male to plus one for female, then the gender associated with man might be minus one, for woman might be plus one and then eventually, learning these things maybe for king you get minus 0.95, for queen plus 0.97, and for apple and orange sort of genderless. Another feature might be, well how royal are these things and so the terms, man and woman are not really royal, so they might have feature values close to zero. Whereas king and queen are highly royal. And apple and orange are not really loyal. How about age? Well, man and woman doesn't connotes much about age. Maybe men and woman implies that they're adults, but maybe neither necessarily young nor old. So maybe values close to zero. Whereas kings and queens are always almost always adults and apple and orange might be more neutral with respect to age and then, another feature for here, is this is a food? Well, man is not a food, woman is not a food, neither are kings and queens, but apples and oranges are foods. And they can be many other features as well ranging from, what is the size of this? What is the cost? Is this something that is a live? Is this an action, or is this a noun, or is this a verb, or is it something else? And so on. So you can imagine coming up with many features. And for the sake of the illustration let's say, 300 different features, and what that does is, it allows you to take this list of numbers, I've only written four here, but this could be a list of 300 numbers, that then becomes a 300 dimensional vector for representing the word man. And I'm going to use the notation e_{5391} to denote a representation like this. And

similarly, this vector, this 300 dimensional vector or 300 dimensional vector like this, I would denote e9853 to denote a 300 dimensional vector we could use to represent the word woman and similarly, for the other examples here. Now, if you use this representation to represent the words orange and apple, then notice that the representations for orange and apple are now quite similar. Some of the features will differ because of the color of an orange, the color an apple, the taste, or some of the features would differ. But by a large, a lot of the features of apple and orange are actually the same, or take on very similar values and so, this increases the odds of the learning algorithm that has figured out that orange juice is a thing, to also quickly figure out that apple juice is a thing. So this allows it to generalize better across different words. So over the next few videos, we'll find a way to learn words embeddings. We just need you to learn high dimensional feature vectors like these, that gives a better representation than one-hot vectors for representing different words. And the features we'll end up learning, won't have a easy to interpret interpretation like that component one is gender, component two is royal, component three is age and so on. Exactly what they're representing will be a bit harder to figure out. But nonetheless, the featurized representations we will learn, will allow an algorithm to quickly figure out that apple and orange are more similar than say, king and orange or queen and orange. If we're able to learn a 300 dimensional feature vector or 300 dimensional embedding for each words, one of the popular things to do is also to take this 300 dimensional data and embed it say, in a two dimensional space so that you can visualize them. And so, one common algorithm for doing this is the **t-SNE algorithm** due to Laurens van der Maaten and Geoff Hinton

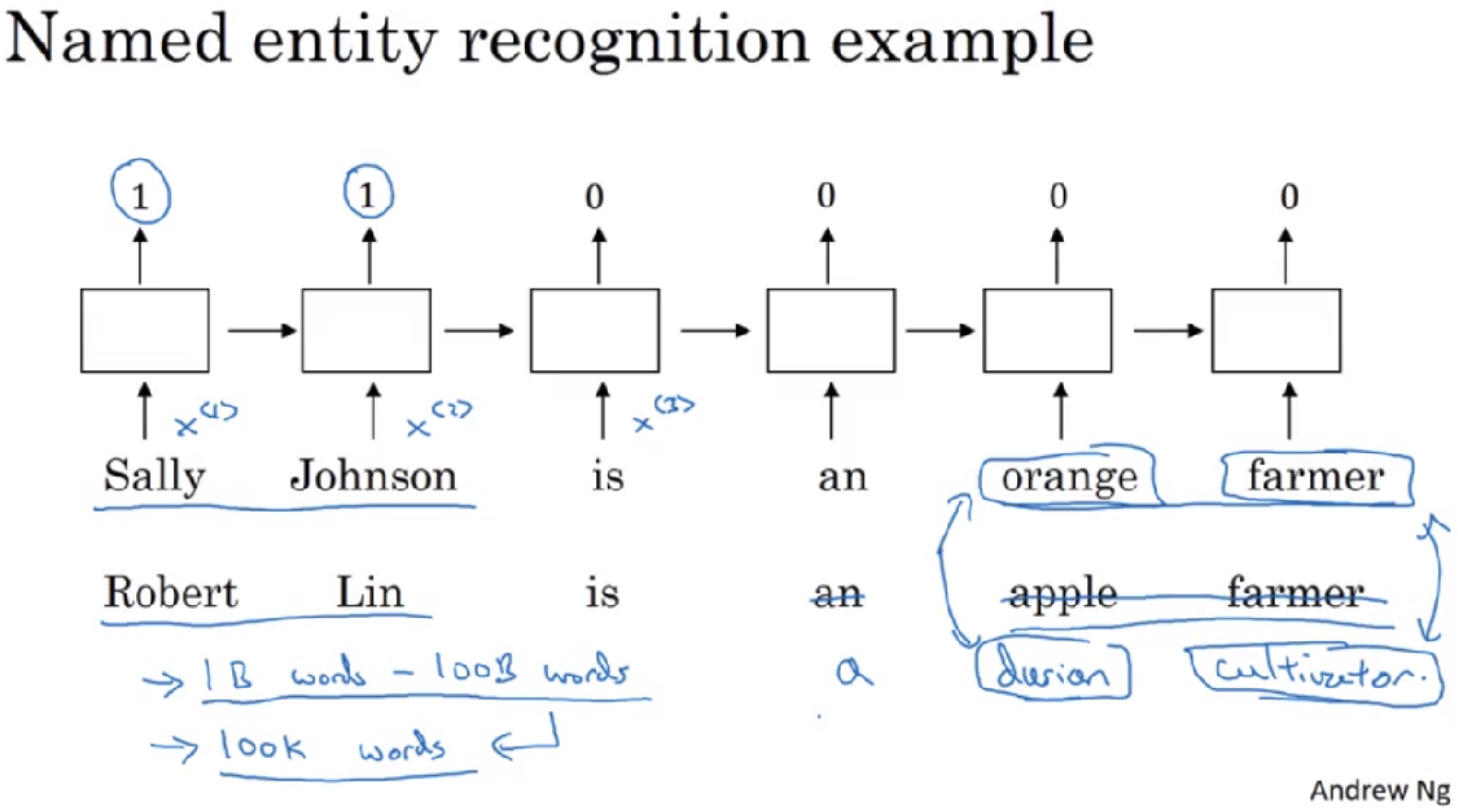


and if you look at one of these embeddings, one of these representations, you find that words like man and woman tend to get grouped together, king and queen tend to get grouped together, and these are the people which tends to get grouped together. Those are animals who can get grouped together. Fruits will tend to be close to each other. Numbers like one, two, three, four, will be close to each other. And then, maybe the animate objects as whole will also tend to be grouped together. But you see plots like these sometimes on the internet to visualize some of these 300 or higher dimensional embeddings and maybe this gives you a sense that, word embeddings algorithms like this can learn similar features for concepts that feel like they should be more related, as visualized by that concept that seem to you and me like they should be more similar, end up getting mapped to a more similar feature vectors and these representations will use these sort of featurized representations in maybe a 300 dimensional space, **these are called embeddings and the reason we call them embeddings is, you can think of a 300 dimensional space and again, they can't draw out here in two dimensional space because it's a 3D one and what you do is you take every words like orange, and have a three dimensional feature vector so that word orange gets embedded to a point in this 300 dimensional space and the word apple, gets**

embedded to a different point in this 300 dimensional space and of course to visualize it, algorithms like **t-SNE**, map this to a much lower dimensional space, you can actually plot the 2D data and look at it. But that's what the term embedding comes from. Word embeddings has been one of the most important ideas in NLP, in Natural Language Processing. In this section, we saw why you might want to learn or use word embeddings. In the next section, let's take a deeper look at how you'll be able to use these algorithms, to build NLP algorithims.

Using word embeddings

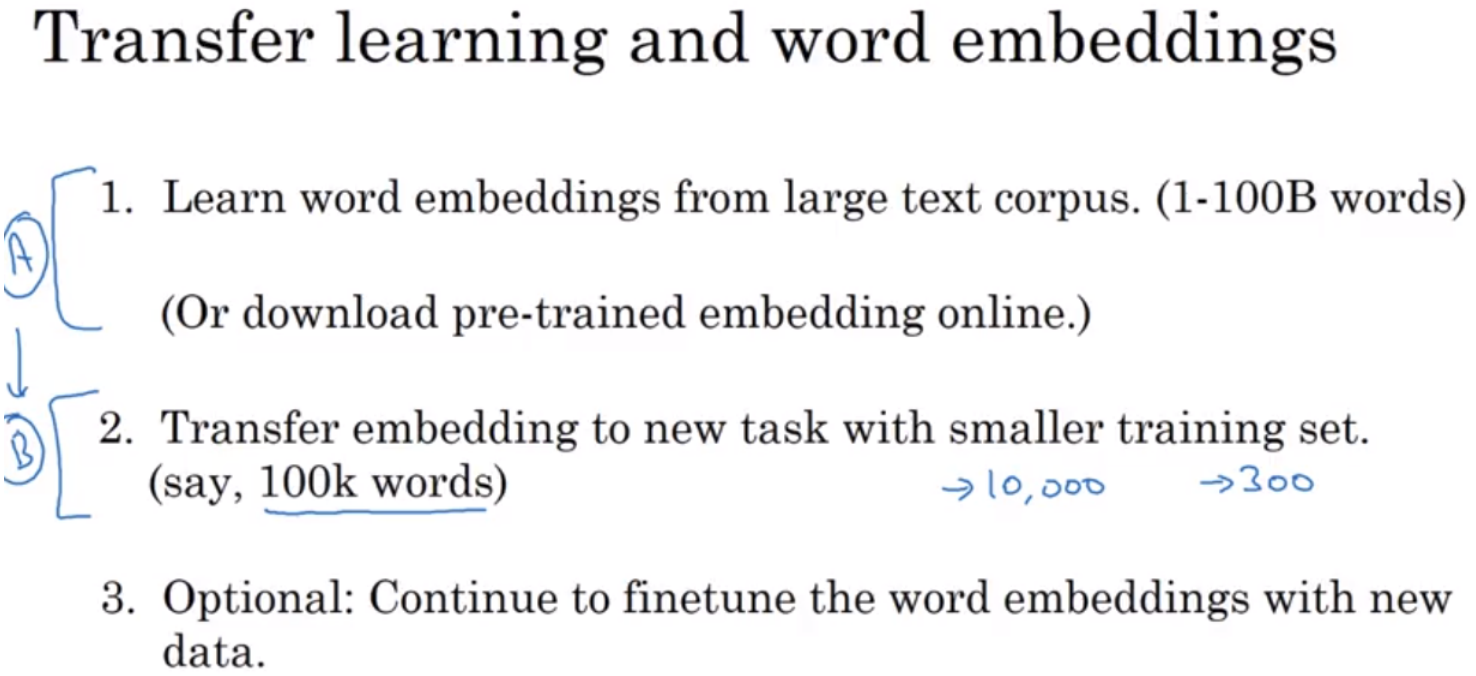
In the last section, we saw what it might mean to learn a featurized representations of different words. In this section, we'll see how we can take these representations and plug them into NLP applications. Let's start with an example.



Continuing with the named entity recognition example, if you're trying to detect people's names. Given a sentence like Sally Johnson is an orange farmer, hopefully, you'll figure out that Sally Johnson is a person's name, hence, the outputs 1 like that. And one way to be sure that Sally Johnson has to be a person, rather than say the name of the corporation is that you know orange farmer is a person. So previously, we had talked about one hot representations to represent these words, x(1), x(2), and so on. But if you can now use the featurized representations, the embedding vectors that we talked about in the last video. Then after having trained a model that uses word embeddings as the inputs, if you now see a new input, Robert Lin is an apple farmer. Knowing that orange and apple are very similar will make it easier for your learning algorithm to generalize to figure out that Robert Lin is also a human, is also a person's name. One of the most interesting cases will be, what if in your test set you see not Robert Lin is an apple farmer, but you see much less common words? What if you see Robert Lin is a durian cultivator? A durian is a rare type of fruit, popular in Singapore and a few other countries. But if you have a small label training set for the **named entity recognition** task, you might not even have seen the word durian or seen the word cultivator in your training set. I guess technically, this should be a durian cultivator. But if you have learned a word embedding that tells you that durian is a fruit, so it's like an orange, and a cultivator, someone that cultivates is like a farmer, then you might still be generalize from having seen an orange farmer in your training set to knowing that a durian cultivator is also probably a person. So one of the reasons that word embeddings will be able to do this is the algorithms to learning word embeddings can examine very large text corpuses, maybe found off the Internet. So you can examine very large data sets, maybe a billion words, maybe even up to 100 billion words would be quite reasonable. So very large training sets of just

unlabeled text and by examining tons of unlabeled text, which you can download more or less for free, you can figure out that orange and durian are similar and farmer and cultivator are similar, and therefore, learn embeddings, that groups them together. Now having discovered that orange and durian are both fruits by reading massive amounts of Internet text, what you can do is then take this word embedding and apply it to your named entity recognition task, for which you might have a much smaller training set, maybe just 100,000 words in your training set, or even much smaller and so this allows you to carry out transfer learning, where you take information you've learned from huge amounts of unlabeled text that you can suck down essentially for free off the Internet to figure out that orange, apple, and durian are fruits and then transfer that knowledge to a task, such as named entity recognition, for which you may have a relatively small labeled training set and, of course, for simplicity, l drew this for it only as a unidirectional RNN. If you actually want to carry out the named entity recognition task, you should, of course, use a bidirectional RNN rather than a simpler one I've drawn here.

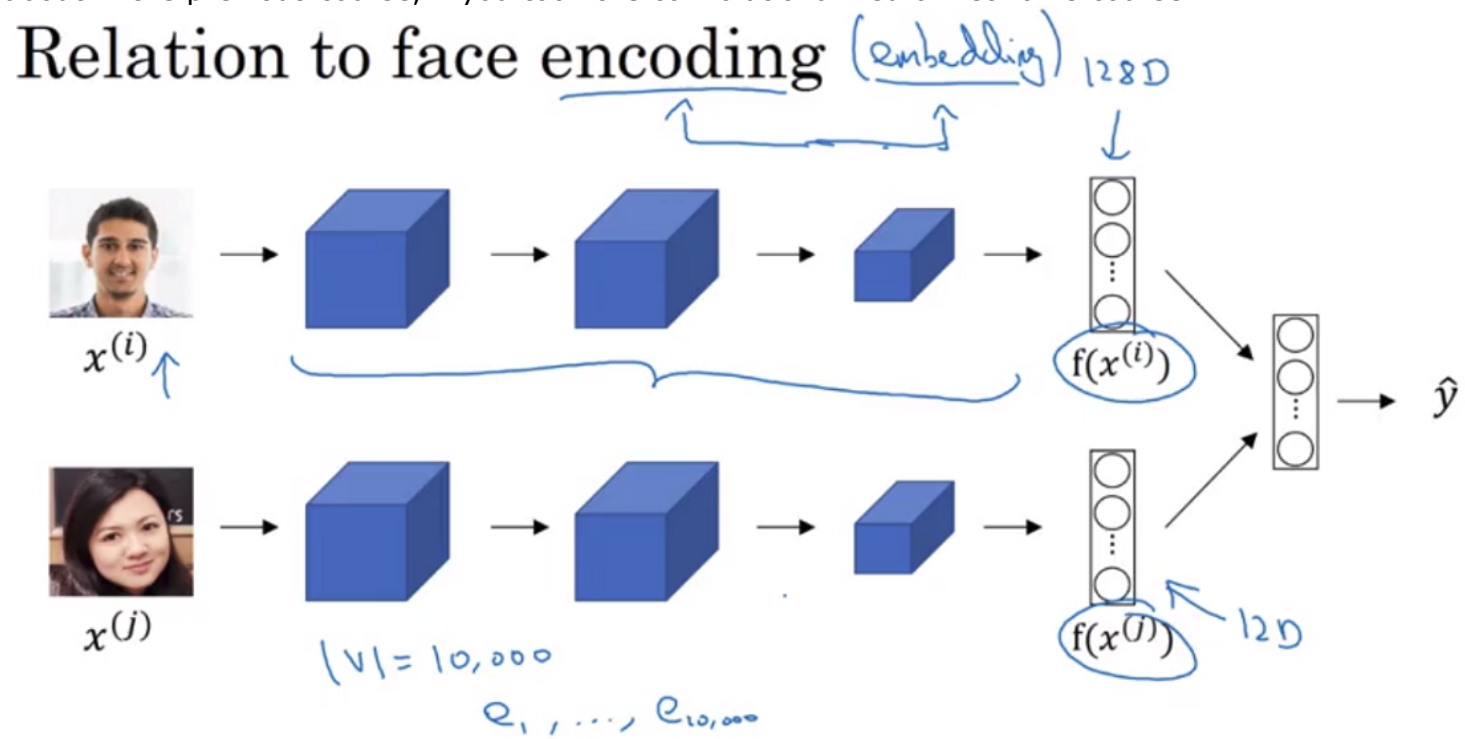
But to summarize, this is how you can carry out transfer learning using word embeddings.



Step 1 is to learn word embeddings from a large text corpus, a very large text corpus or you can also download pre-trained word embeddings online. There are several word embeddings that you can find online under very permissive licenses and you can then take these word embeddings and transfer the embedding to new task, where you have a much smaller labeled training sets and use this, let's say, 300 dimensional embedding, to represent your words. One nice thing also about this is you can now use relatively lower dimensional feature vectors. So rather than using a 10,000 dimensional one-hot vector, you can now instead use maybe a 300 dimensional dense vector. Although the one-hot vector is fast and the 300 dimensional vector that you might learn for your embedding will be a dense vector and then, finally, as you train your model on your new task, on your named entity recognition task with a smaller label data set, one thing you can optionally do is to continue to fine tune, continue to adjust the word embeddings with the new data. In practice, you would do this only if this task 2 has a pretty big data set. If your label data set for step 2 is quite small, then usually, I would not bother to continue to fine tune the word embeddings. So word embeddings tend to make the biggest difference when the task you're trying to carry out has a relatively smaller training set. So it has been useful for many NLP tasks and I'll just name a few. It has been useful for named entity recognition, for text summarization, for co-reference resolution, for parsing. These are all maybe pretty standard NLP tasks. It has been less useful for language modeling, machine translation, especially if you're accessing a language modeling or machine translation task for which you have a lot of data just dedicated to that task. So as seen in other transfer learning settings, if you're trying to transfer from some task A to some task B, the process of transfer learning is just most useful when you happen to

have a ton of data for A and a relatively smaller data set for B. And so that's true for a lot of NLP tasks, and just less true for some language modeling and machine translation settings.

Finally, word embeddings has a interesting relationship to the face encoding ideas that you learned about in the previous course, if you took the convolutional neural networks course.



So you will remember that for face recognition, we train this Siamese network architecture that would learn, say, a 128 dimensional representation for different faces. And then you can compare these encodings in order to figure out if these two pictures are of the same face. The words encoding and embedding mean fairly similar things. So in the face recognition literature, people also use the term encoding to refer to these vectors, f(x(i)) and f(x(j)). One difference between the face recognition literature and what we do in word embeddings is that, for face recognition, you wanted to train a neural network that can take as input any face picture, even a picture you've never seen before, and have a neural network compute an encoding for that new picture. Whereas what we'll do, and you'll understand this better when we go through the next few videos, whereas what we'll do for learning word embeddings is that we'll have a fixed vocabulary of, say, 10,000 words. And we'll learn a vector e_1 through, say, $e_{10,000}$ that just learns a fixed encoding or learns a fixed embedding for each of the words in our vocabulary. So that's one difference between the set of ideas you saw for face recognition versus what the algorithms we'll discuss in the next few videos. But the terms encoding and embedding are used somewhat interchangeably. So the difference I just described is not represented by the difference in terminologies. It's just a difference in how we need to use these algorithms in face recognition, where there's unlimited sea of pictures you could see in the future. Versus natural language processing, where there might be just a fixed vocabulary, and everything else like that we'll just declare as an unknown word.

So in this section, you saw how using word embeddings allows you to implement this type of transfer learning. And how, by replacing the one-hot vectors we're using previously with the embedding vectors, you can allow your algorithms to generalize much better, or you can learn from much less label data. Next, I want to show you just a few more properties of these word embeddings and then after that, we will talk about algorithms for actually learning these word embeddings.

Properties of word embeddings

From the previous sections we have seen how word embeddings can help in building NLP applications. One of the most fascinating properties of word embeddings is that they can also help with analogy reasoning and while reasonable analogies may not be by itself the most important NLP application, they might also help convey a sense of what these word embeddings are doing, what these word embeddings can do.

Let see a diagram where we have a featurized representations of a set of words that you might hope a word embedding could capture.

Analogies

	Man (5391)	Woman (9853)	King (4914)	Queen (7157)	Apple (456)	Orange (6257)
Gender	−1	1	-0.95	0.97	0.00	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.00
Age	0.03	0.02	0.70	0.69	0.03	-0.02
Food	0.09	0.01	0.02	0.01	0.95	0.97

e_{5391} e_{man} e_{woman}

$e_{man} - e_{woman} \approx \begin{bmatrix} -2 \\ 0 \\ 0 \\ 0 \end{bmatrix}$

Man → Woman as King → ? Queen

$e_{king} - e_{queen} \approx \begin{bmatrix} -2 \\ 0 \\ 0 \\ 0 \end{bmatrix}$

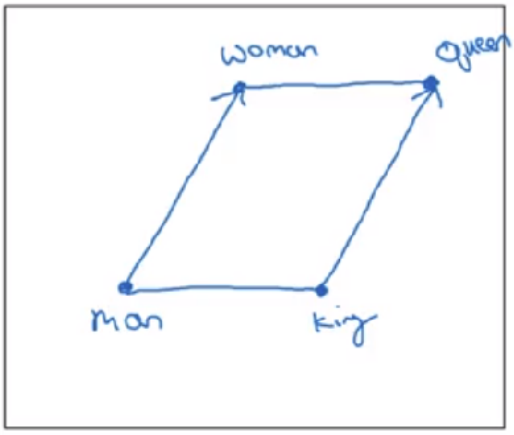
$e_{man} - e_{woman} \approx e_{king} - e_?$ e_{queen}

[Mikolov et. al., 2013, Linguistic regularities in continuous space word representations]

Andre

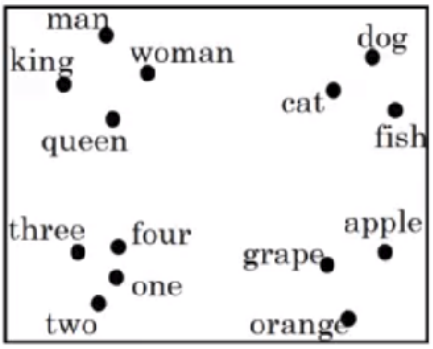
Let's say I pose a question, man is to woman as king is to what? Many of you will say, man is to woman as king is to queen. But is it possible to have an algorithm figure this out automatically?

Analogies using word vectors



300D

3000 → 2D



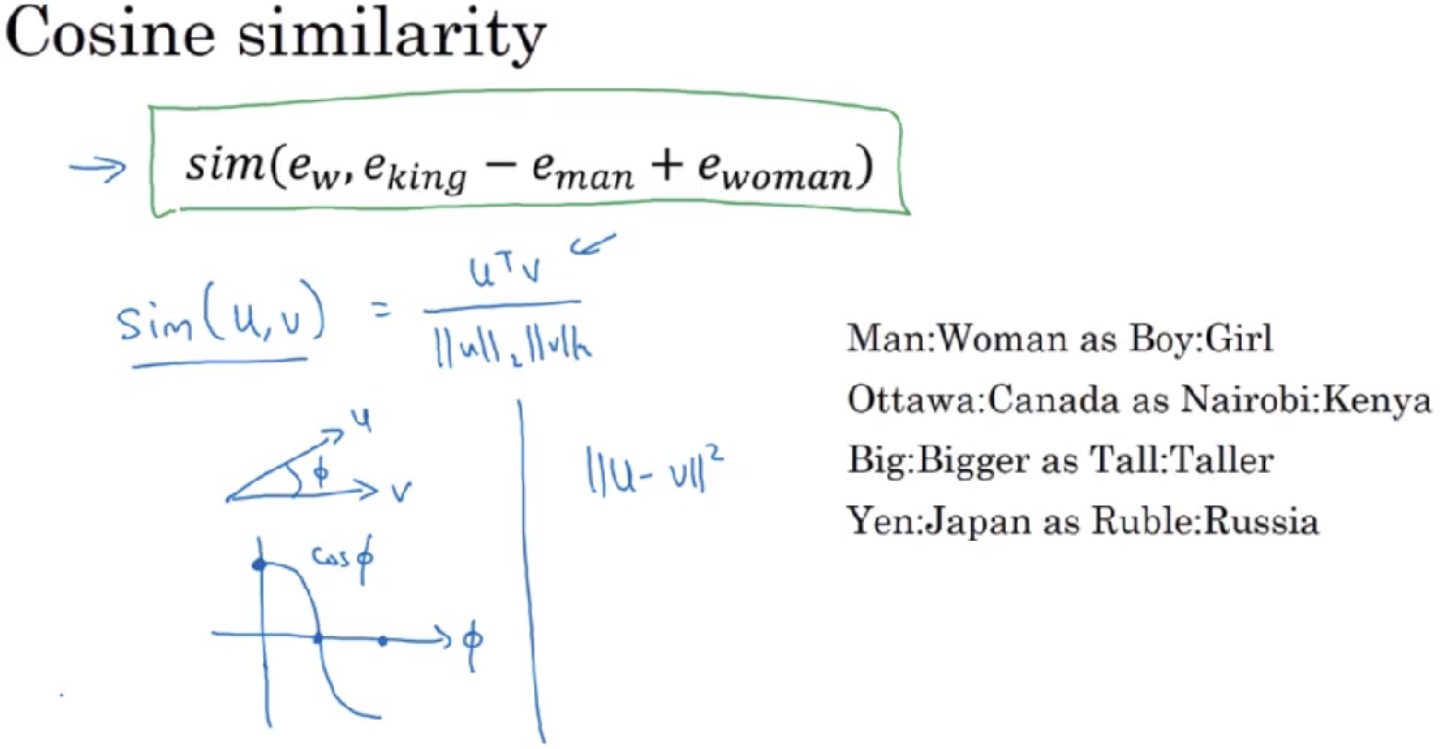
t-SNE

$e_{man} - e_{woman} \approx e_{king} - e_?$ e_w

Find word w: $\arg\max_w$ $\text{sim}(e_w, e_{king} - e_{man} + e_{woman})$

30 - 75%

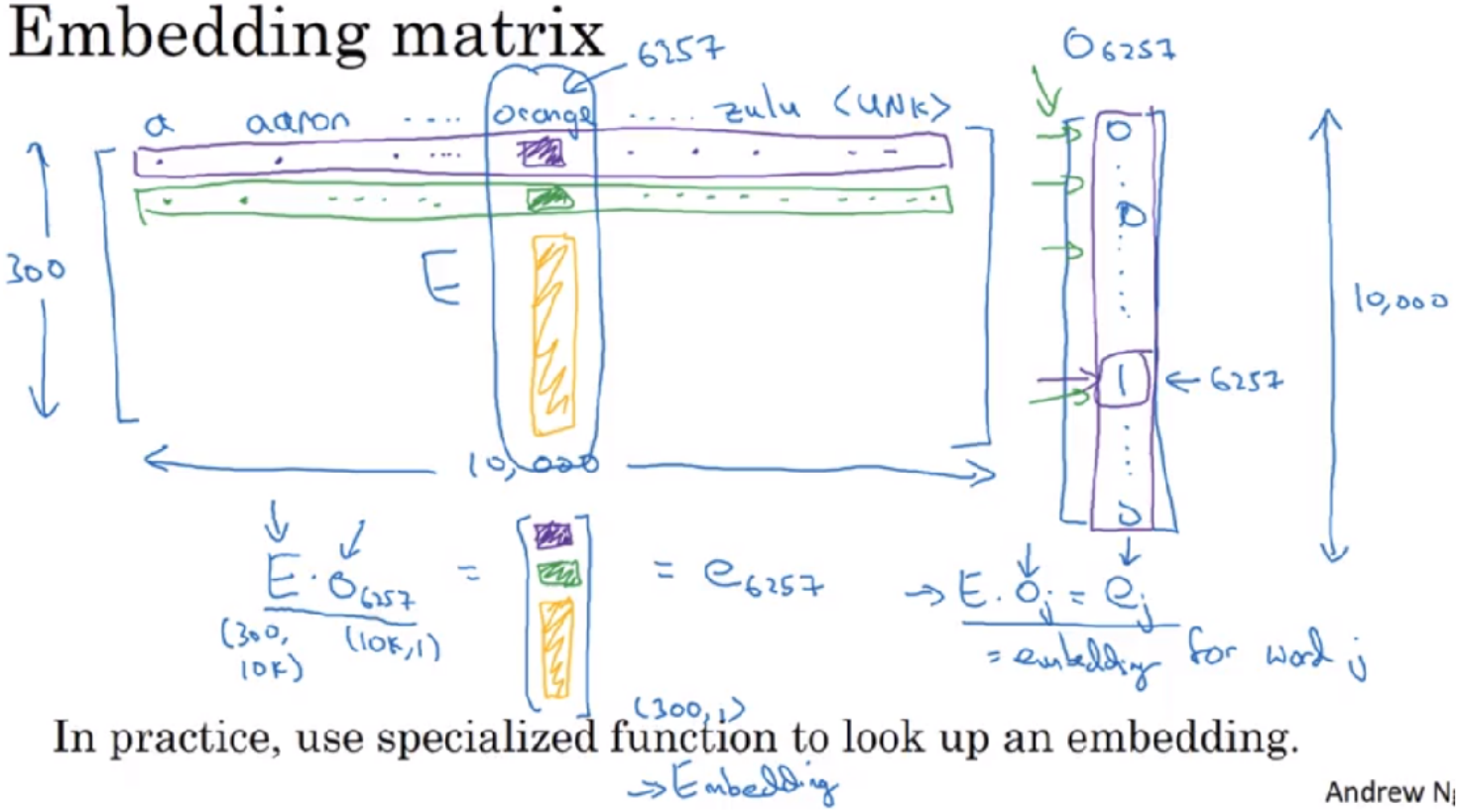
Andrew



So one of the remarkable results about word embeddings is the generality of analogy relationships they can learn. So for example, it can learn that man is to woman as boy is to girl, because the vector difference between man and woman, similar to king and queen and boy and girl, is primarily just the gender. It can learn that Ottawa, which is the capital of Canada, that Ottawa is to Canada as Nairobi is to Kenya. So that's the city capital is to the name of the country. It can learn that big is to bigger as tall is to taller, and it can learn things like that. Yen is to Japan, since yen is the currency of Japan, as ruble is to Russia and all of these things can be learned just by running a word embedding learning algorithm on the large text corpus. It can spot all of these patterns by itself, just by running from very large bodies of text. So in this section, we saw how word embeddings can be used for analogy reasoning and while you might not be trying to build an analogy reasoning system yourself as an application, this I hope conveys some intuition about the types of feature-like representations that these representations can learn. And you also saw how cosine similarity can be a way to measure the similarity between two different word embeddings. Now, we talked a lot about properties of these embeddings and how you can use them. Next, let's talk about how you'd actually learn these word embeddings.

Embedding matrix

Let's start to formalize the problem of learning a good word embedding. When you implement an algorithm to learn a word embedding, what you end up learning is an embedding matrix. Let's take a look at what that means. Let's say, as usual we're using our 10,000-word vocabulary. So, the vocabulary has A, Aaron, Orange, Zulu, maybe also unknown word as a token. What we're going to do is learn embedding matrix E, which is going to be a 300 dimensional by 10,000 dimensional matrix, if you have 10,000 words vocabulary or maybe 10,001 is our word token, there's one extra token and the columns of this matrix would be the different embeddings for the 10,000 different words you have in your vocabulary.



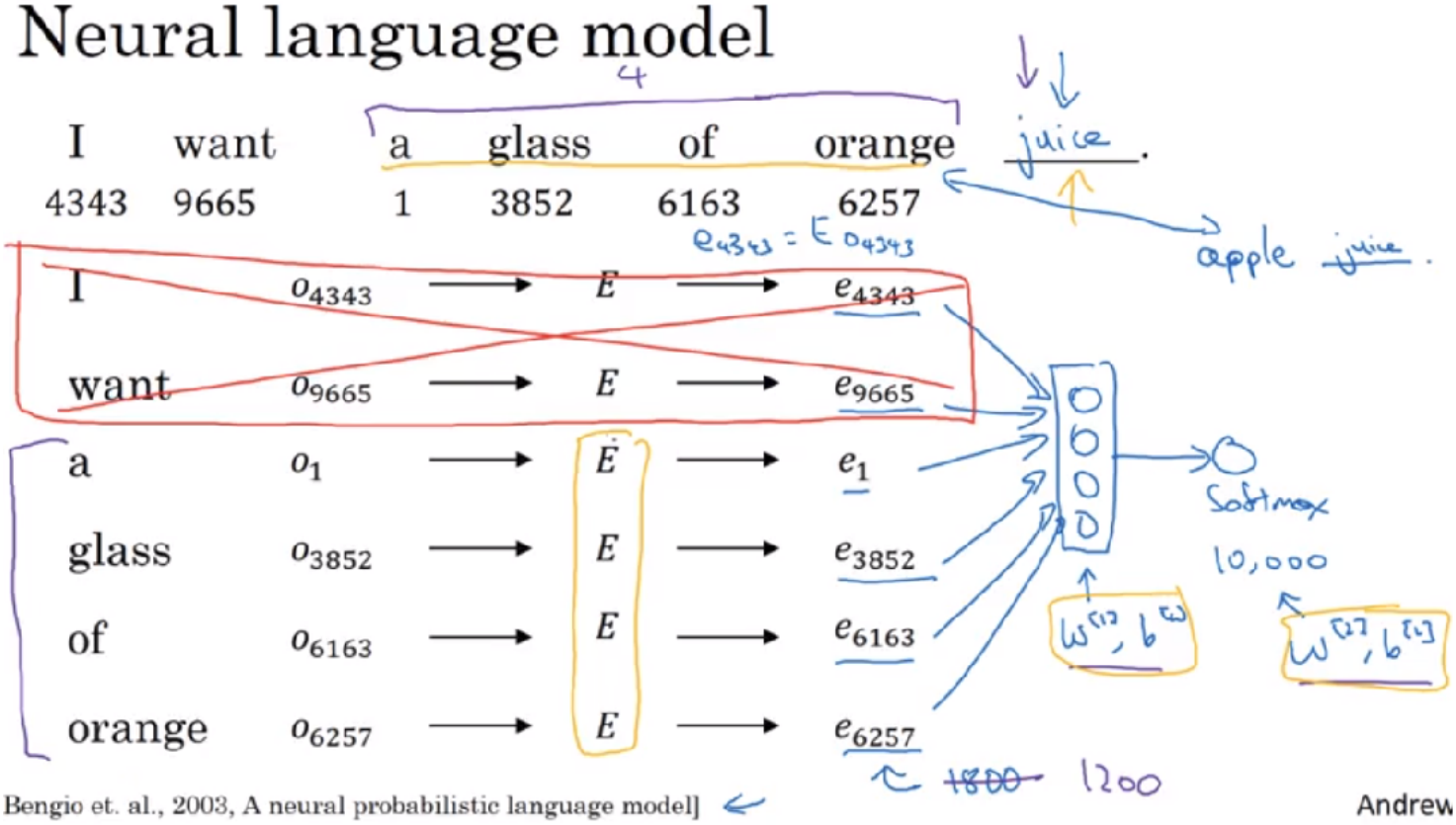
So, Orange was word number 6257 in our vocabulary of 10,000 words. So, one piece of notation we'll use is that O_{6257} was the one-hot vector with zeros everywhere and a one in position 6257. And so, this will be a 10,000-dimensional vector with a one in just one position. So, this isn't quite a drawn scale. Yes, this should be as tall as the embedding matrix on the left is wide. And if the embedding matrix is called capital E then notice that if you take E and multiply it by just one-hot vector by 0 of 6257, then this will be a 300-dimensional vector. So, E is 300x10,000 and O is 10,000x1. So, the product will be 300x1, so with 300-dimensional vector and notice that to compute the first element of this vector, of this 300-dimensional vector, what you do is you will multiply the first row of the matrix E with this. But all of these elements are zero except for element 6257 and so you end up with zero times this, zero times this, zero times this, and so on and then, 1 times whatever this is, and zero times this, zero times this, zero times and so on and so, you end up with the first element as whatever is that elements up there, under the Orange column and then, for the second element of this 300-dimensional vector we're computing, you would take the vector 6257 and multiply it by the second row with the matrix E. So again, you have zero times this, plus zero times this, plus zero times all of these are the elements and then one times this, and then zero times everything else and add that together. So you end up with this and so on as you go down the rest of this column. So, that's why the embedding matrix E times this one-hot vector here winds up selecting out this 300-dimensional column corresponding to the word Orange. So, this is going to be equal to e_{6257} which is the notation we're going to use to represent the embedding vector that 300 by one dimensional vector for the word Orange. And more generally, E for a specific word W, this is going to be embedding for a word W and more generally, E times O substitute J, one-hot vector with one that position J, this is going to be E_J and that's going to be the embedding for word J in the vocabulary. So, the thing to remember from this slide is that our goal will be to learn an embedding matrix E and what you see in the next section is you initialize E randomly and you're straight in the sense to learn all the parameters of this 300 by 10,000 dimensional matrix and E times this one-hot vector gives you the embedding vector. Now just one note, when we're writing the equation, it'll be convenient to write this type of notation where you take the matrix E and multiply it by the one-hot vector O. But if when you're implementing this, it is not efficient to actually implement this as a mass matrix vector multiplication because the one-hot vectors, now this is a relatively high dimensional vector and most of these elements are zero. So, it's actually not efficient to use a matrix vector multiplication to implement this because if we multiply a whole bunch of things by zeros and so the practice, you would actually use a specialized function to just look up a column of the Matrix E rather than do this

with the matrix multiplication. But writing of the map, it is just convenient to write it out this way. So, **in Keras for example there is a embedding layer and we use the embedding layer then it more efficiently just pulls out the column you want from the embedding matrix rather than does it with a much slower matrix vector multiplication**. So, in this section you saw the notations were used to describe algorithms to learning these embeddings and the key terminology is this matrix capital E which contain all the embeddings for the words of the vocabulary. In the next section, we'll start to talk about specific algorithms for learning this matrix E.

Learning Word Embeddings: Word2vec & GloVe

Learning word embeddings

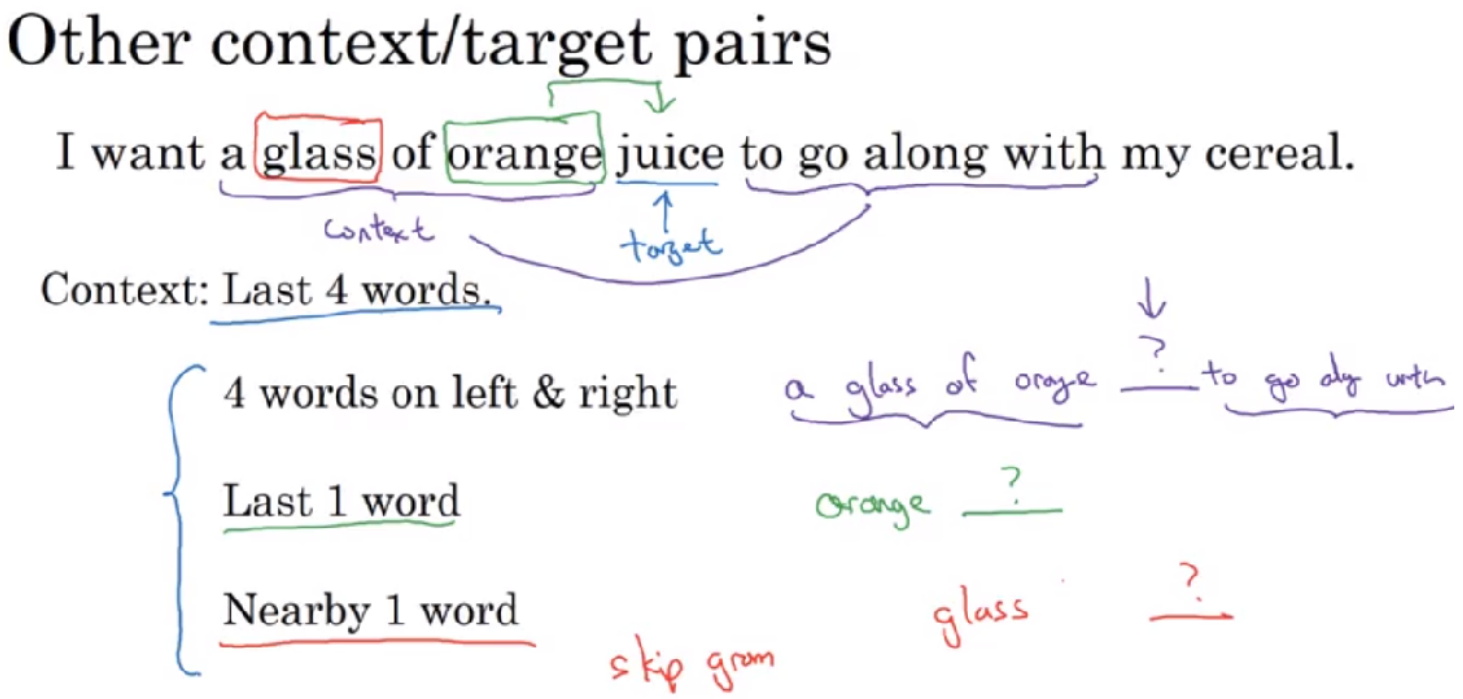
In this section, you'll start to learn some concrete algorithms for learning word embeddings. In the history of deep learning as applied to learning word embeddings, people actually started off with relatively complex algorithms and then over time, researchers discovered they can use simpler and simpler and simpler algorithms and still get very good results especially for a large dataset. But what happened is, some of the algorithms that are most popular today, they are so simple that if I present them first, it might seem almost a little bit magical, how can something this simple work? So, what I'm going to do is start off with some of the slightly more complex algorithms because I think it's actually easier to develop intuition about why they should work, and then we'll move on to simplify these algorithms and show you some of the simple algorithms that also give very good results. So, let's get started. Let's say you're building a language model and you do it with a neural network. So, during training, you might want your neural network to do something like input, I want a glass of orange, and then predict the next word in the sequence and below each of these words, I have also written down the index in the vocabulary of the different words. So it turns out that building a neural language model is the small way to learn a set of embeddings and the ideas I present on this slide were due to Yoshua Bengio, Rejean Ducharme, Pascals Vincent, and Christian Jauvin. So, here's how you can build a neural network to predict the next word in the sequence.



Let me take the list of words, I want a glass of orange, and let's start with the first word I. So I'm going to construct one add vector corresponding to the word I. So there's a one add vector with a one in position, 4343. So this is going to be 10,000 dimensional vector and what we're going to do is then have a matrix of parameters E, and take E times O to get an embedding vector e4343, and this step really means that e4343 is obtained by the matrix E times the one hot vector 43 and then we'll do the same for all of the other words. So the word want, is where 9665 one add vector, multiply by E

to get the embedding vector and similarly, for all the other words. A, is a first word in dictionary, alphabetic comes first, so there is O one, gets this E one. And similarly, for the other words in this phrase. So now you have a bunch of three dimensional embedding, so each of this is a 300 dimensional embedding vector. and what we can do, is feed all of them into a neural network. So here is the neural network layer. And then this neural network feeds to a softmax, which has it's own parameters as well. And a softmax classifies among the 10,000 possible outputs in the vocab for those final word we're trying to predict. And so, if in the training slide we saw the word juice then, the target for the softmax in training repeat that it should predict the other word juice was what came after this. So this hidden name here will have his own parameters. So have some, I'm going to call this W1 and there's also B1. The softmax there was this own parameters W2, B2, and they're using 300 dimensional word embeddings, then here we have six words. So, this would be 6x300. So this layer or this input will be a 1,800 dimensional vector obtained by taking your six embedding vectors and stacking them together. Well, what's actually more commonly done is to have a fixed historical window. So for example, you might decide that you always want to predict the next word given say the previous four words, where four here is a hyperparameter of the algorithm. So this is how you adjust to either very long or very short sentences or you decide to always just look at the previous four words, so you say, I will still use those four words and so, if you're always using a four word history, this means that your neural network will input a 1,200 dimensional feature vector, go into this layer, then have a softmax and try to predict the output. And again, variety of choices. And using a fixed history, just means that you can deal with even arbitrarily long sentences because the input sizes are always fixed. So, the parameters of this model will be this matrix E, and use the same matrix E for all the words. So you don't have different matrices for different positions in the proceedings four words, is the same matrix E and then, these weights are also parameters of the algorithm and you can use that crop to perform gradient to sent to maximize the likelihood of your training set to just repeatedly predict given four words in a sequence, what is the next word in your text corpus? And it turns out that this algorithm we'll learn pretty decent word embeddings and the reason is, if you remember our orange juice, apple juice example, is in the algorithm's incentive to learn pretty similar word embeddings for orange and apple because doing so allows it to fit the training set better because it's going to see orange juice sometimes, or see apple juice sometimes, and so, if you have only a 300 dimensional feature vector to represent all of these words, the algorithm will find that it fits the training set fast. If apples, oranges, and grapes, and pears, and so on and maybe also durians which is a very rare fruit and that with similar feature vectors. **So, this is one of the earlier and pretty successful algorithms for learning word embeddings, for learning this matrix E**.

But now let's generalize this algorithm and see how we can derive even simpler algorithms. So, I want to illustrate the other algorithms using a more complex sentence as our example. Let's say that in your training set, you have this longer sentence, I want a glass of orange juice to go along with my cereal. So, what we saw on the last slide was that the job of the algorithm was to predict some word juice, which we are going to call the target words, and it was given some context which was the last four words. And so, if your goal is to learn a embedding of researchers I've experimented with many different types of context. If it goes to build a language model then is natural for the context to be a few words right before the target word. But if your goal is into learn the language model per se, then you can choose other contexts. For example, you can pose a learning problem where the context is the four words on the left and right. So, you can take the four words on the left and right as the context, and what that means is that we're posing a learning problem where the algorithm is given four words on the left. So, a glass of orange, and four words on the right, to go along with, and this has to predict the word in the middle. And posing a learning problem like this where you have the embeddings of the left four words and the right four words feed into a neural network, similar to what you saw in the previous slide, to try to predict the word in the middle, try to put it target word in the middle, this can also be used to learn word embeddings. Or if you want to use a simpler context, maybe you'll just use the last one word. So given just the word orange, what comes after orange? So this will be different learning problem where you tell it one word, orange, and will say well, what do you think is the next word and you can construct a neural network that just fits in the word, the one previous word or the embedding of the one previous word to a neural network as you try to predict the next word. Or, one thing that works surprisingly well is to take a nearby one word.



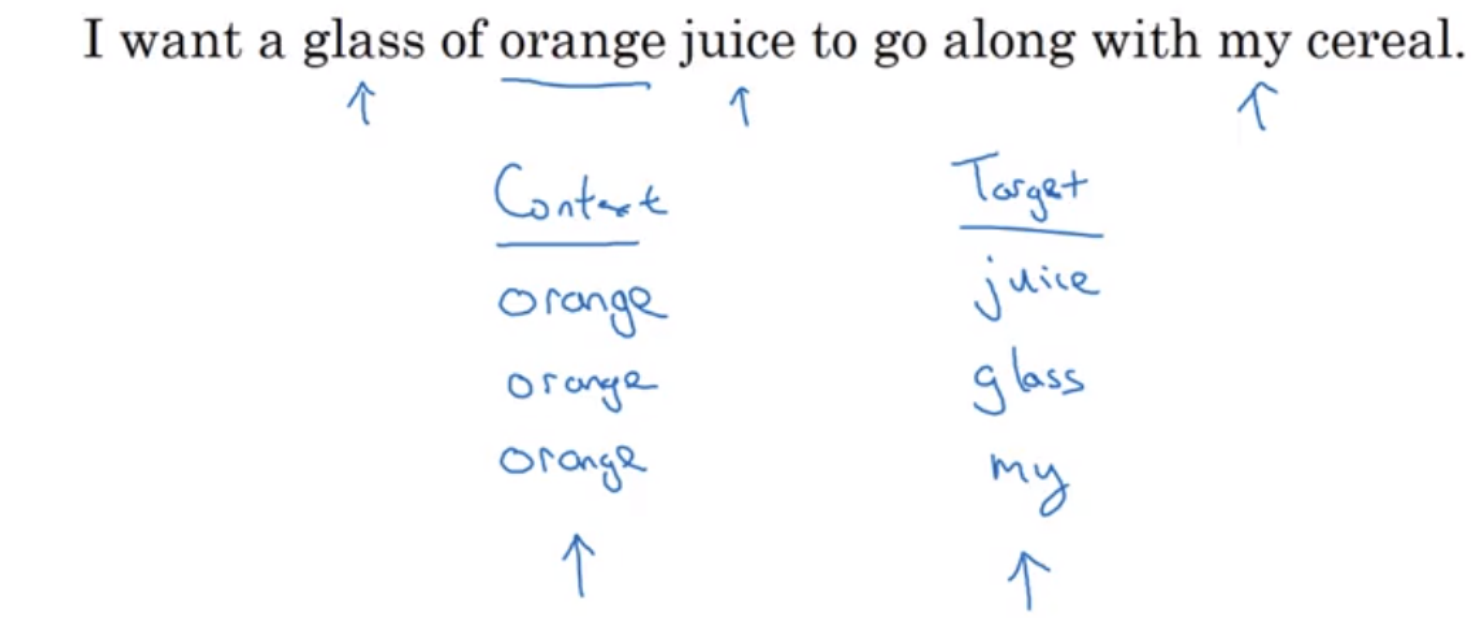
Some might tell you that, well, take the word glass, is somewhere close by. Some might say, I saw the word glass and then there's another words somewhere close to glass, what do you think that word is? So, that'll be using nearby one word as the context and we'll formalize this in the next section but this is the idea of a **Skip-Gram model**, and just an example of a simpler algorithm where the context is now much simpler, is just one word rather than four words, but this works remarkably well. So what researchers found was that if you really want to build a language model, it's natural to use the last few words as a context. But if your main goal is really to learn a word embedding, then you can use all of these other contexts and they will result in very meaningful work embeddings as well. I will formalize the details of this in the next section where we talk about the Word2Vec model.

To summarize, in this section you saw how the language modeling problem which causes the pose of machines learning problem where you input the context like the last four words and predicts some target words, how posing that problem allows you to learn input word embedding. In the next section, you'll see how using even simpler context and even simpler learning algorithms to mark from context to target word, can also allow you to learn a good word embedding.

Word2Vec

In the last section, we saw how you can learn a natural language model in order to get good word embeddings. In this section, you see the Word2Vec algorithm which is simple and comfortably more efficient way to learn this types of embeddings. Lets take a look. Most of the ideas I'll present in this section are due to Tomas Mikolov, Kai Chen, Greg Corrado, and Jeff Dean. Let's say you're given this sentence in your training set. In the skip-gram model, what we're going to do is come up with a few context to target errors to create our supervised learning problem. So rather than having the context be always the last four words or the last end words immediately before the target word, what we're going to do is, say, randomly pick a word to be the context word and let's say we choose the word orange and what we're going to do is randomly pick another word within some window. Say +-5 words or +-10 words of the context word and we choose that to be target word. So maybe just by chance you might pick juice to be a target word, that's just one word later. Or you might choose two words before. So you have another pair where the target could be glass or, maybe just by chance you choose the word my as the target and so we'll set up a supervised learning problem where given the context word, you're asked to predict what is a randomly chosen word within say, a +-10 word window, or +-5 or 10 word window of that input context word and obviously, this is not a very easy learning problem, because within +- 10 words of the word orange, it could be a lot of different words. But a goal that's setting up this supervised learning problem, isn't to do well on the supervised learning problem per se, it is that we want to use this learning problem to learn good word embeddings.

Skip-grams



Mikolov et. al., 2013. Efficient estimation of word representations in vector space.]

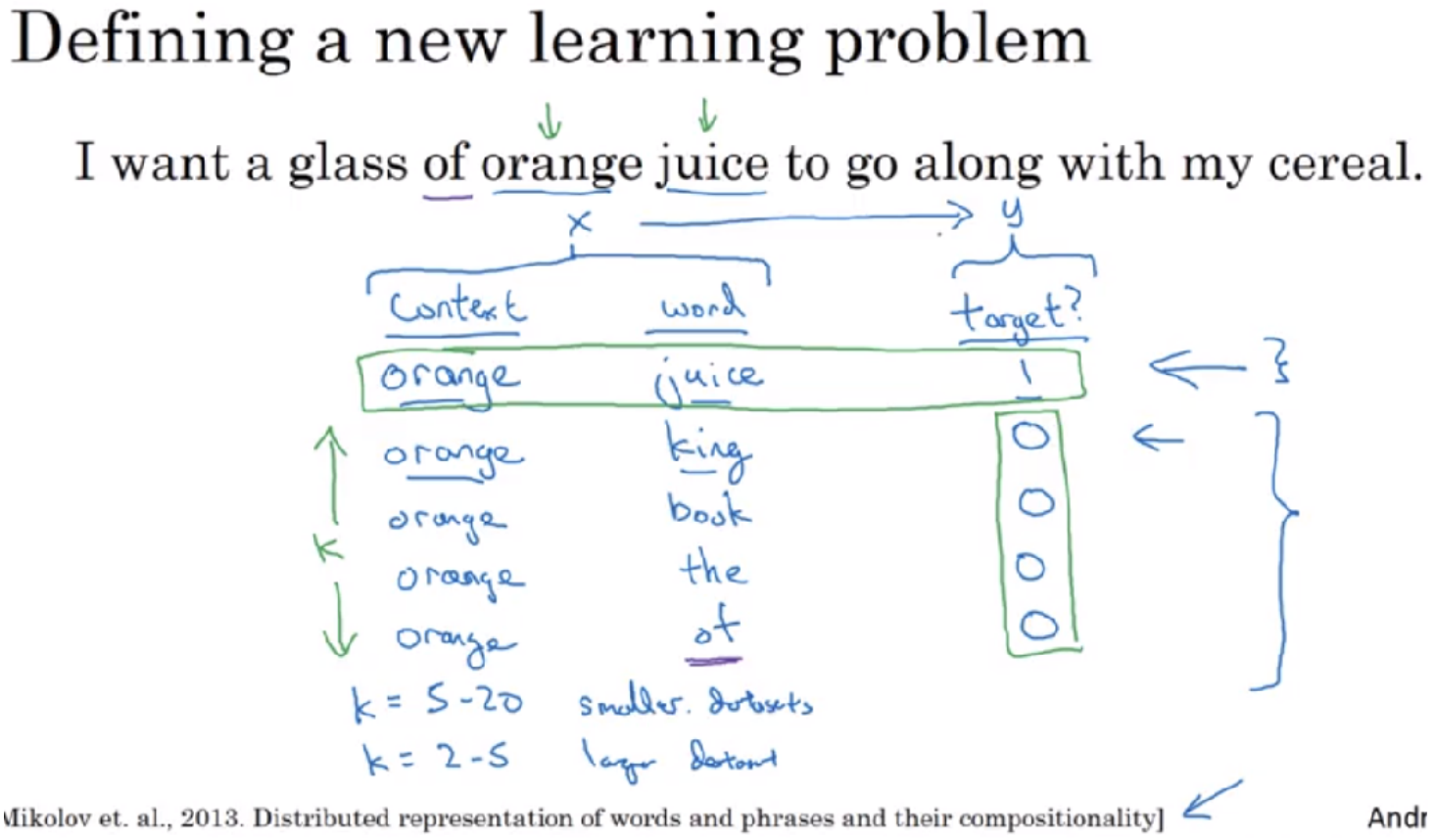
So, here are the details of the model. Let's say that we'll continue to our vocab of 10,000 words and some have been on vocab sizes that exceeds a million words but the basic supervised learning problem we're going to solve is that we want to learn the mapping from some Context c, such as the word orange to some target, which we will call t, which might be the word juice or the word glass or the word my, if we use the example from the previous section. So in our vocabulary, orange is word 6257, and the word
juice is the word 4834 in our vocab of 10,000 words and so that's the input x that you want to learn to map to that open y. So to represent the input such as the word orange, we can start out with some one hot vector which is going to be write as O subscript C, so there's a one hot vector for the context words and then similar to what we saw on the last section we can take the embedding matrix E, multiply E by the vector O subscript C, and this gives you your embedding vector for the input context word, so here EC is equal to capital E times that one hot vector. Then in this neural network that we formed we're going to take this vector EC and feed it to a softmax unit. So I've been drawing softmax unit as a node in a neural network. That's not an o, that's a softmax unit and then there's a drop in the softmax unit to output y hat.
So to write out this model in detail. This is the model, the softmax model, probability of different tanka words given the input context word as e to the e, theta t transpose, ec. Divided by some over all words, so we're going to say, sum from J equals one to all 10,000 words of e to the theta j transposed ec. So here theta T is the parameter associated with, I'll put t, but really there's a chance of a particular word, t, being the label.
So I've left off the biased term to solve mass but we could include that too if we wish.
And then finally the loss function for softmax will be the usual.
So we use y to represent the target word. And we use a one-hot representation for y hat and y here. Then the lost would be The negative log liklihood, so sum from i equals 1 to 10,000 of yi log yi hat. So that's a usual loss for softmax where we're representing the target y as a one hot vector. So this would be a one hot vector with just 1 1 and the rest zeros. And if the target word is juice, then it'd be element 4834 from up here. That is equal to 1 and the rest will be equal to 0. And similarly Y hat will

be a 10,000 dimensional vector output by the softmax unit with probabilities for all 10,000 possible targets words. So to summarize, this is the overall little model, little neural network with basically looking up the embedding and then just a soft max unit. And the matrix E will have a lot of parameters, so the matrix E has parameters corresponding to all of these embedding vectors, E subscript C. And then the softmax unit also has parameters that gives the theta T parameters but if you optimize this loss function with respect to the all of these parameters, you actually get a pretty good set of embedding vectors. So this is called the skip-gram model because is taking as input one word like orange and then trying to predict some words skipping a few words from the left or the right side. To predict what comes little bit before little bit after the context words. Now, it turns out there are a couple problems with using this algorithm. And the primary problem is computational speed. In particular, for the softmax model, every time you want to evaluate this probability, you need to carry out a sum over all 10,000 words in your vocabulary. And maybe 10,000 isn't too bad, but if you're using a vocabulary of size 100,000 or a 1,000,000, it gets really slow to sum up over this denominator every single time. And, in fact, 10,000 is actually already that will be quite slow, but it makes even harder to scale to larger vocabularies. So there are a few solutions to this, one which you see in the literature is to use a hierarchical softmax classifier. And what that means is, instead of trying to categorize something into all 10,000 carries on one go. Imagine if you have one classifier, it tells you is the target word in the first 5,000 words in the vocabulary? Or is in the second 5,000 words in the vocabulary? And lets say this binary cost that it tells you this is in the first 5,000 words, think of second class to tell you that this in the first 2,500 words of vocab or in the second 2,500 words vocab and so on. Until eventually you get down to classify exactly what word it is, so that the leaf of this tree, and so having a tree of classifiers like this, means that each of the retriever nodes of the tree can be just a binding classifier. And so you don't need to sum over all 10,000 words or else it will capsize in order to make a single classification. In fact, the computational classifying tree like this scales like log of the vocab size rather than linear in vocab size. So this is called a hierarchical softmax classifier. I should mention in practice, the hierarchical softmax classifier doesn't use a perfectly balanced tree or this perfectly symmetric tree, with equal numbers of words on the left and right sides of each branch. In practice, the hierarchical software classifier can be developed so that the common words tend to be on top, whereas the less common words like durian can be buried much deeper in the tree. Because you see the more common words more often, and so you might need only a few traversals to get to common words like the and of. Whereas you see less frequent words like durian much less often, so it says okay that are buried deep in the tree because you don't need to go that deep. So there are various heuristics for building the tree how you used to build the hierarchical software spire.
So this is one idea you see in the literature, the speeding up the softmax classification.
But I won't spend too much more time.
And you can read more details of this on the paper that I referenced by Thomas
and others, on the first slide. But I won't spend too much more time on this. Because in the next video, where she talk about a different method, called nectar sampling, which I think is even simpler. And also works really well for speeding up the softmax classifier and the problem of needing the sum over the entire cap size in the denominator. So you see more of that in the next video. But before moving on, one quick Topic I want you to understand is how to sample the context C. So once you sample the context C, the target T can be sampled within, say, a plus minus ten word window of the context C, but how do you choose the context C? One thing you could do is just sample uniformly, at random, from your training corpus. When we do that, you find that there are some words like the, of, a, and, to and so on that appear extremely frequently. And so, if you do that, you find that in your context to target mapping pairs just get these these types of words extremely frequently, whereas there are other words like orange, apple, and also durian that don't appear that often. And maybe you don't want your training site to be dominated by these extremely frequently or current words, because then you spend almost all the effort updating ec, for those frequently occurring words. But you want to make sure that you spend some time updating the embedding, even for these less common words like e durian. So in practice the distribution of words pc isn't taken just entirely uniformly at random for the training set purpose, but instead there are different heuristics that you could use in order to balance out something from the common words together with the less common words.

So that's it for the Word2Vec skip-gram model. If you read the original paper by that I referenced earlier, you find that that paper actually had two versions of this Word2Vec model, the skip gram was one. And the other one is called the CBow, the continuous backwards model, which takes the surrounding contexts from middle word, and uses the surrounding words to try to predict the middle word, and that algorithm also works, it has some advantages and disadvantages. But the key problem with this algorithm with the skip-gram model as presented so far is that the softmax step is very expensive to calculate because needing to sum over your entire vocabulary size into the denominator of the soft packs. In the next video I show you an algorithm that modifies the training objective that makes it run much more efficiently therefore lets you apply this in a much bigger fitting set as well and therefore learn much better word embeddings.

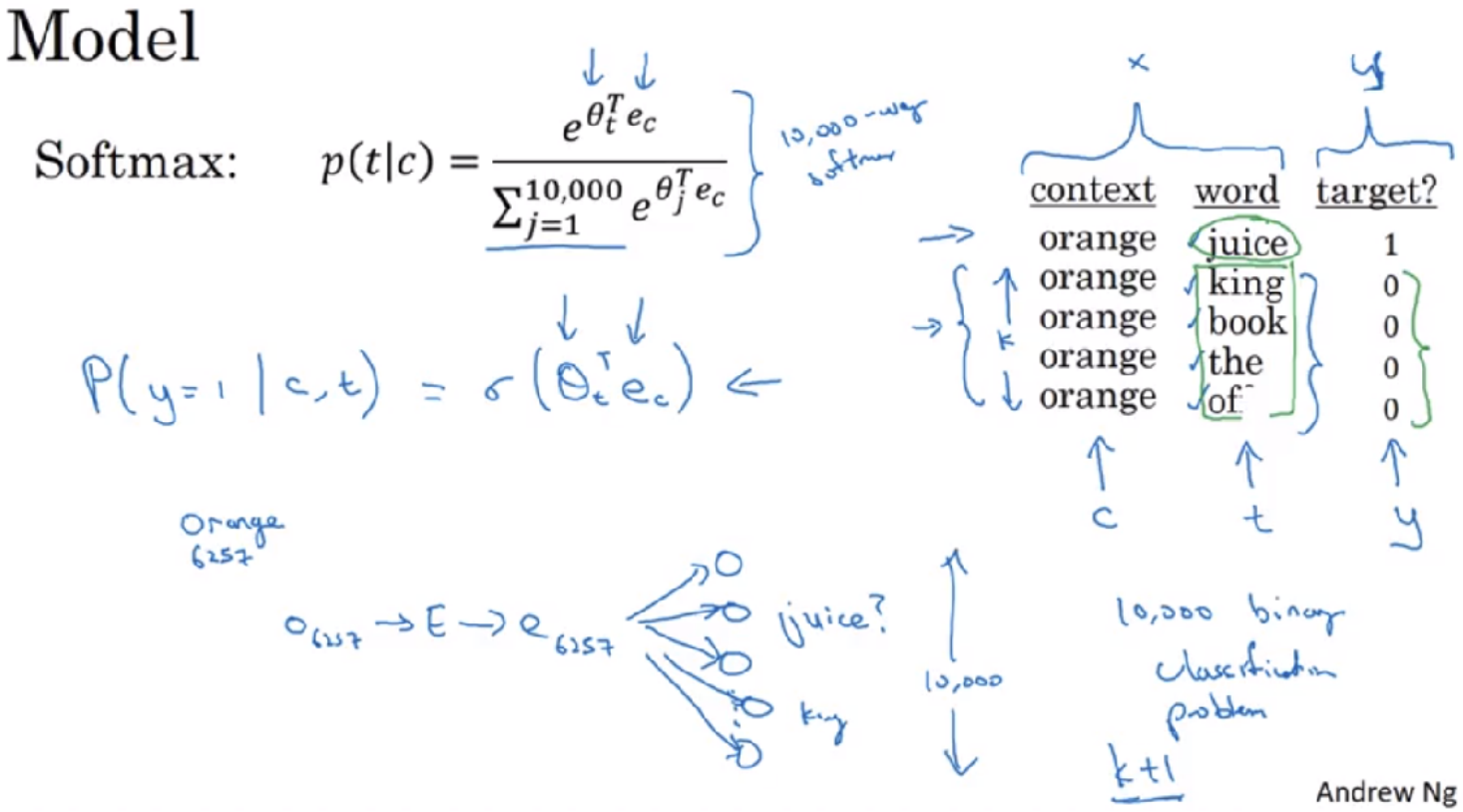
Negative Sampling

In the last section, we saw how the Skip-Gram model allows us to construct a supervised learning task. So we map from context to target and how that allows you to learn a useful word embedding. But the downside of that was the Softmax objective was slow to compute. In this section, we'll see a modified learning problem called **negative sampling** that allows us to do something similar to the **Skip-Gram model** we saw just now, but with a much more efficient learning algorithm. Let's see how we can do this. Most of the ideas presented in this video are due to Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeff Dean. So what we're going to do in this algorithm is create a new supervised learning problem and the problem is, given a pair of words like orange and juice, we're going to predict, is this a context-target pair? So in this example, orange juice was a positive example. And how about orange and king? Well, that's a negative example, so we're going to write 0 for the target. So what we're going to do is we're actually going to sample a context and a target word. So in this case, we have orange and juice and we'll associate that with a label of 1, so just put words in the middle and then having generated a positive example, so the positive example is generated exactly how we generated it in the previous section. Sample a context word, look around a window of say, plus-minus ten words and pick a target word. So that's how we generate the first row of this table with orange, juice, 1 and then to generate a negative example, we're going to take the same context word and then just pick a word at random from the dictionary.



So in this case, we chose the word king at random and we will label that as 0 and then let's take orange and let's pick another random word from the dictionary. Under the assumption that if we pick

a random word, it probably won't be associated with the word orange, so orange, book, 0 and let's pick a few others, orange, maybe just by chance, we'll pick the 0 and then orange and then orange, and maybe just by chance, we'll pick the word of and we'll put a 0 there and notice that all of these are labeled as 0 even though the word of actually appears next to orange as well. So to summarize, the way we generated this data set is, we'll pick a context word and then pick a target word and that is the first row of this table. That gives us a positive example. So context, target, and then give that a label of 1 and then what we'll do is for some number of times say, k times, we're going to take the same context word and then pick random words from the dictionary, king, book, the, of, whatever comes out at random from the dictionary and label all those 0, and those will be our negative examples and it's okay if just by chance, one of those words we picked at random from the dictionary happens to appear in the window, in a plus-minus ten word window say, next to the context word, orange. Then we're going to create a supervised learning problem where the learning algorithm inputs x, inputs this pair of words, and it has to predict the target label to predict the output y. So the problem is really given a pair of words like orange and juice, do you think they appear together? Do you think I got these two words by sampling two words close to each other? Or do you think I got them as one word from the text and one word chosen at random from the dictionary? It's really to try to distinguish between these two types of distributions from which you might sample a pair of words. So this is how you generate the training set. How do you choose k, Mikolov et al, recommend that maybe **k is 5 to 20** for smaller data sets and if you have a very large data set, then chose k to be smaller. So k equals 2 to 5 for larger data sets, and large values of k for smaller data sets. Okay, and in this example, we'll just use k = 4. Next, let's describe the supervised learning model for learning a mapping from x to y. So here was the Softmax model you saw from the previous section and here is the training set we got from the previous section where again, this is going to be the new input x and this is going to be the value of y you're trying to predict.



So to define the model, I'm going to use this to denote, this was c for the context word, this to denote the possible target word, t, and this, I'll use y to denote 0, 1, this is a context target pair. So what we're going to do is define a logistic regression model. Say, that the chance of y = 1, given the input c, t pair, we're going to model this as basically a regression model, but the specific formula we'll use s sigma applied to theta transpose, theta t transpose, e c. So the parameters are similar as before, you have one parameter vector theta for each possible target word. And a separate parameter vector, really the embedding vector, for each possible context word. And we're going to use this formula to estimate the probability that y is equal to 1. So if you have k examples here, then you can think of this as having a k to 1 ratio of negative to positive examples. So for every positive

examples, you have k negative examples with which to train this logistic regression-like model and so to draw this as a neural network, if the input word is orange, which is word 6257, then what you do is, you input the one hot vector passing through e, do the multiplication to get the embedding vector 6257 and then what you have is really 10,000 possible logistic regression classification problems. Where one of these will be the classifier corresponding to, well, is the target word juice or not? And then there will be other words, for example, there might be ones somewhere down here which is predicting, is the word king or not and so on, for these possible words in your vocabulary. So think of this as having 10,000 binary logistic regression classifiers, but instead of training all 10,000 of them on every iteration, we're only going to train five of them. We're going to train the one responding to the actual target word we got and then train four randomly chosen negative examples. And this is for the case where k is equal to 4. So instead of having one giant 10,000 way Softmax, which is very expensive to compute, we've instead turned it into 10,000 binary classification problems, each of which is quite cheap to compute. And on every iteration, we're only going to train five of them or more generally, k + 1 of them, of k negative examples and one positive examples. And this is why the computation cost of this algorithm is much lower because you're updating k + 1, let's just say units, k + 1 binary classification problems. Which is relatively cheap to do on every iteration rather than updating a 10,000 way Softmax classifier. So you get to play with this algorithm in the problem exercise for this week as well. S**o this technique is called negative sampling because what you're doing is, you have a positive example, the orange and then juice. And then you will go and deliberately generate a bunch of negative examples, negative samplings,** hence, the name negative sampling, with which to train four more of these binary classifiers and on every iteration, you choose four different random negative words with which to train your algorithm on. Now, before wrapping up, one more important detail with this algorithm is, how do you choose the negative examples? So after having chosen the context word orange, how do you sample these words to generate the negative examples? So one thing you could do is sample the words in the middle, the candidate target words. One thing you could do is sample it according to the empirical frequency of words in your corpus. So just sample it according to how often different words appears. But the problem with that is that you end up with a very high representation of words like the, of, and, and so on. One other extreme would be to say, you use 1 over the vocab size, sample the negative examples uniformly at random, but that's also very non-representative of the distribution of English words. So the authors, Mikolov et al, reported that empirically, what they found to work best was to take this heuristic value, which is a little bit in between the two extremes of sampling from the empirical frequencies, meaning from whatever's the observed distribution in English text to the uniform distribution. And what they did was they sampled proportional to their frequency of a word to the power of three-fourths. So if f of wi is the observed frequency of a particular word in the English language or in your training set corpus, then by taking it to the power of three-fourths, this is somewhere in-between the extreme of taking uniform distribution. And the other extreme of just taking whatever was the observed distribution in your training set and so'm not sure this is very theoretically justified, but multiple researchers are now using this heuristic, and it seems to work decently well.

So to summarize, we've seen how you can learn word vectors in a Softmax classier, but it's very computationally expensive and in this section, we saw how by changing that to a bunch of binary classification problems, you can very efficiently learn words vectors and if we run this algorithm, we'll be able to learn pretty good word vectors. Now of course, as is the case in other areas of deep learning as well, there are open source implementations and there are also pre-trained word vectors that others have trained and released online under permissive licenses.

GloVe word vectors

We learned about several algorithms for computing words embeddings. Another algorithm that has some momentum in the NLP community is the GloVe algorithm. This is not used as much as the Word2Vec or the skip-gram models, but it has some enthusiasts. Because I think, in part of its simplicity. Let's take a look. The **GloVe algorithm** was created by Jeffrey Pennington, Richard Socher, and Chris Manning and **GloVe stands for global vectors for word representation**. I think one confusing part of this algorithm is, if you look at equation below

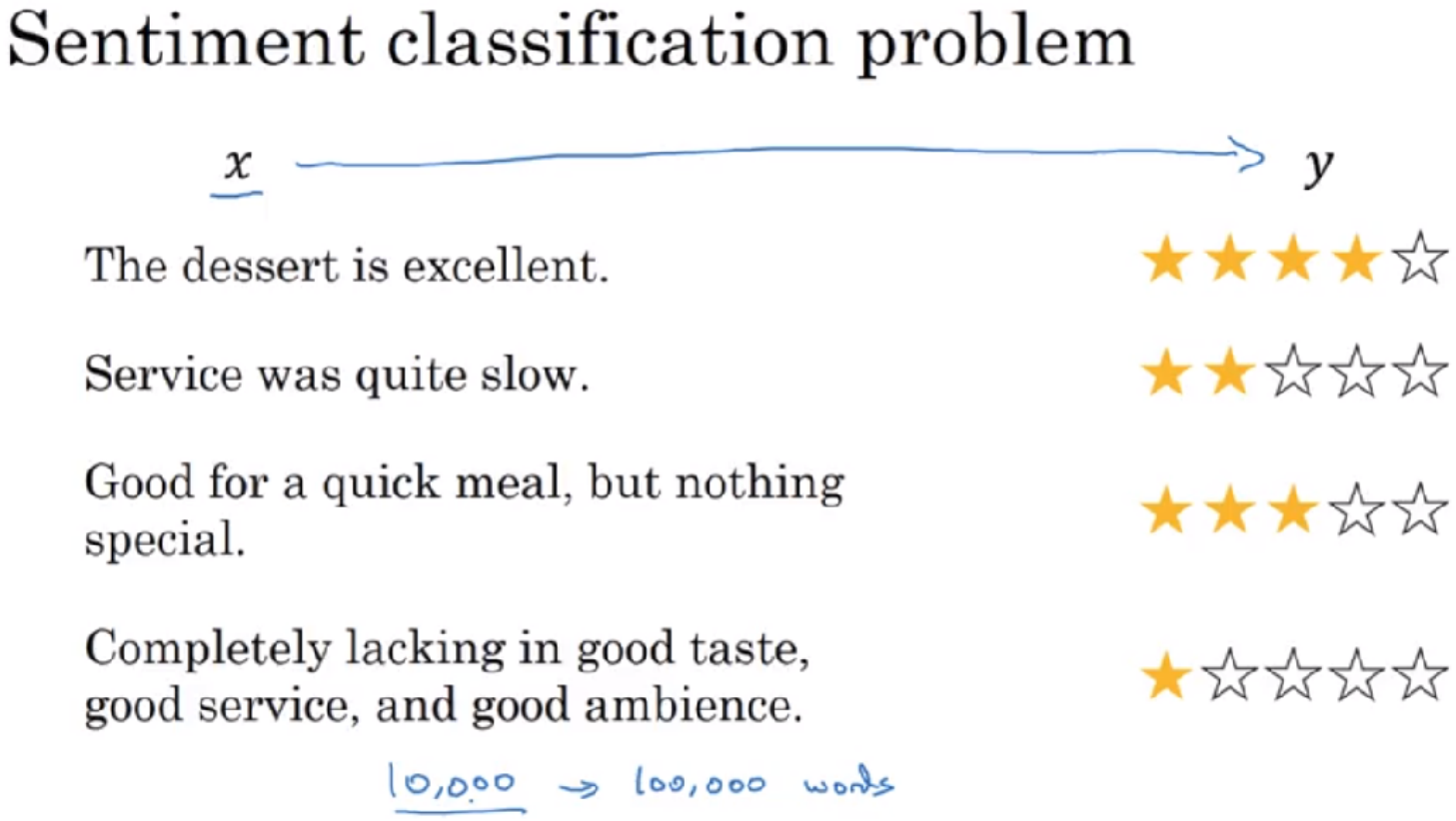
$$\text{minimize} \sum_{i=1}^{10,000} \sum_{j=1}^{10,000} f(X_{ij})(\theta_i^T e_j + b_i - b_j' - \log X_{ij})^2$$

It seems almost too simple. How could it be that just minimizing a **square cost function** like this allows you to learn meaningful word embeddings? But it turns out that this works and the way that the inventors end up with this algorithm was, they were building on the history of much more complicated algorithms like the newer language model, and then later, there came the **Word2Vec skip-gram model**, and then this came later. And we really hope to simplify all of the earlier algorithms. Before concluding our discussion of algorithms concerning word embeddings, there's one more property of them that we should discuss briefly. Which is that? We started off with this featurization view as the motivation for learning word vectors. We said, "Well, maybe the first component of the embedding vector to represent gender, the second component to represent how royal it is, then the age and then whether it's a food, and so on." But when you learn a word embedding using one of the algorithms that we've seen, such as the GloVe algorithm that we just saw, what happens is, you cannot guarantee that the individual components of the embeddings are interpretable. So, that's it for learning word embeddings. We've now seen a variety of algorithms for learning these word embeddings. Next, we'll show how we can use these algorithms to carry out sentiment classification.

Applications using Word Embeddings

Sentiment Classification

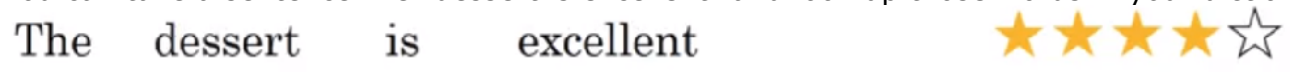
Sentiment classification is the task of looking at a piece of text and telling if someone likes or dislikes the thing they're talking about. It is one of the most important building blocks in NLP and is used in many applications. One of the challenges of sentiment classification is you might not have a huge label training set for it. But with word embeddings, you're able to build good sentiment classifiers even with only modest-size label training sets. Let's see how you can do that. So here's an example of a sentiment classification problem.



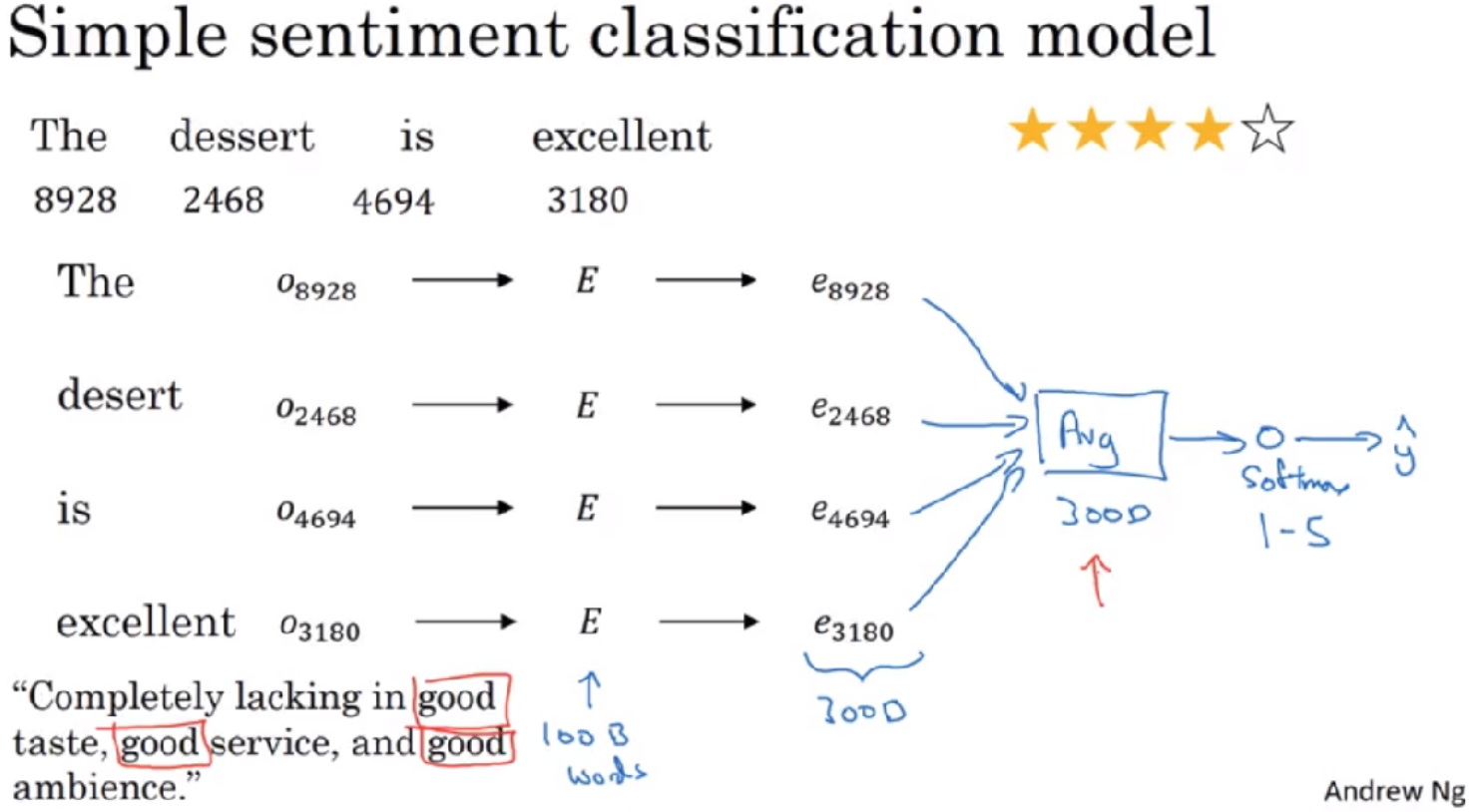
The input X is a piece of text and the output Y that you want to predict is what is the sentiment, such as the star rating of, let's say, a restaurant review. So if someone says, "The dessert is excellent" and they give it a four-star review, "Service was quite slow" two-star review, "Good for a quick meal but nothing special" three-star review. And this is a pretty harsh review, "Completely lacking in good

taste, good service, and good ambiance." That's a one-star review. So if you can train a system to map from X or Y based on a label data set like this, then you could use it to monitor comments that people are saying about maybe a restaurant that you run. So people might also post messages about your restaurant on social media, on Twitter, or Facebook, or Instagram, or other forms of social media. And if you have a sentiment classifier, they can look just a piece of text and figure out how positive or negative is the sentiment of the poster toward your restaurant. Then you can also be able to keep track of whether or not there are any problems or if your restaurant is getting better or worse over time. So one of the challenges of sentiment classification is you might not have a huge label data set. So for sentimental classification task, training sets with maybe anywhere from 10,000 to maybe 100,000 words would not be uncommon. Sometimes even smaller than 10,000 words and word embeddings that you can take can help you to much better understand especially when you have a small training set. So here's what you can do. We'll go for a couple different algorithms in this section. Here's a simple sentiment classification model.

You can take a sentence like "dessert is excellent" and look up those words in your dictionary.

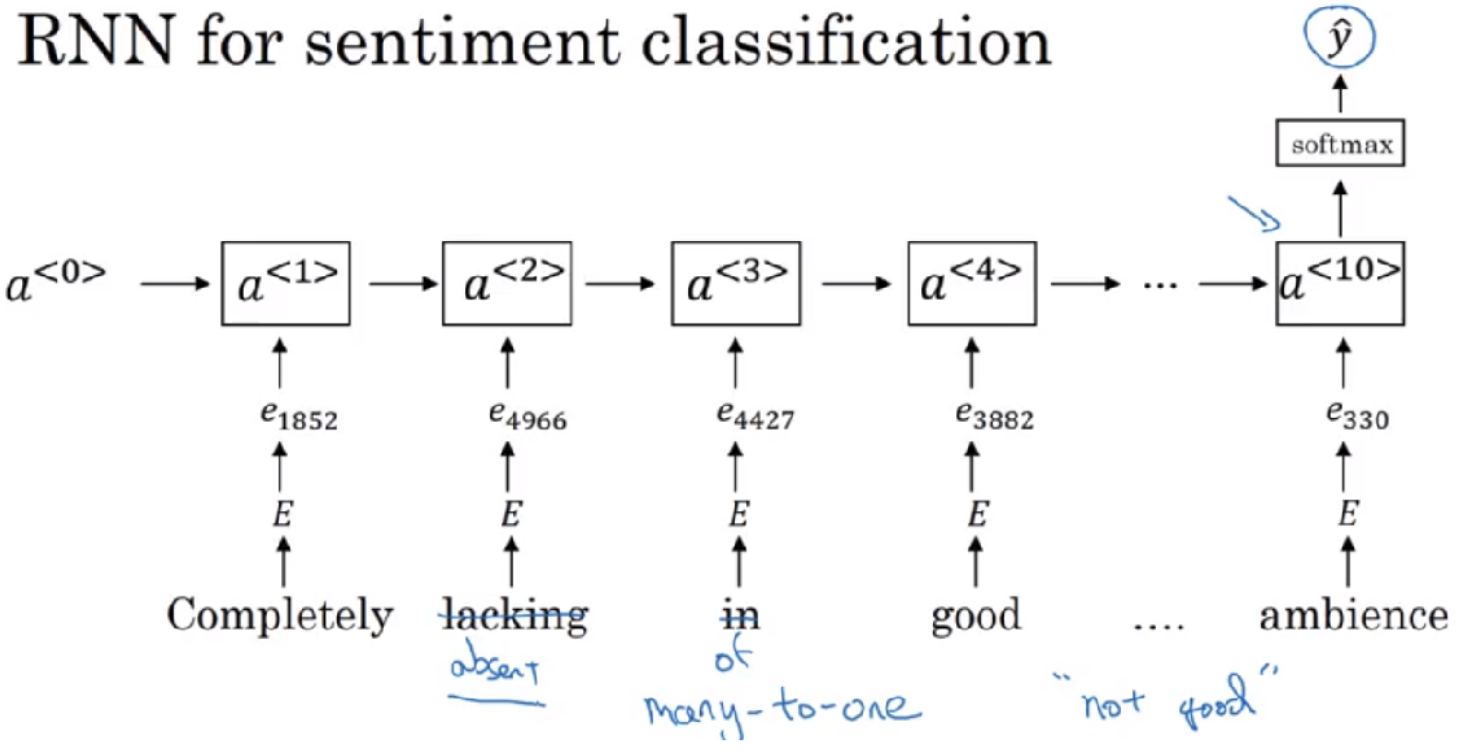


We use a 10,000-word dictionary as usual. And let's build a classifier to map it to the output Y that this was four stars. So given these four words, as usual, we can take these four words and look up the one-hot vector. So there's 0 8 9 2 8 which is a one-hot vector multiplied by the embedding matrix E, which can learn from a much larger text corpus. It can learn in embedding from, say, a billion words or a hundred billion words, and use that to extract out the embedding vector for the word "the", and then do the same for "dessert", do the same for "is" and do the same for "excellent". And if this was trained on a very large data set, like a hundred billion words, then this allows you to take a lot of knowledge even from infrequent words and apply them to your problem, even words that weren't in your labeled training set. Now here's one way to build a classifier, which is that you can take these vectors, let's say these are 300-dimensional vectors, and you could then just sum or average them and I'm just going to put a bigger average operator here and you could use sum or average. And this gives you a 300-dimensional feature vector that you then pass to a soft-max classifier which then outputs Y-hat and so the softmax can output what are the probabilities of the five possible outcomes from one-star up to five-star. So this will be assortment of the five possible outcomes to predict what is Y. So notice that by using the average operation here, this particular algorithm works for reviews that are short or long because even if a review that is 100 words long, you can just sum or average all the feature vectors for all hundred words and so that gives you a representation, a 300-dimensional feature representation, that you can then pass into your sentiment classifier. So this average will work decently well. And what it does is it really averages the meanings of all the words or sums the meaning of all the words in your example. And this will work to [inaudible]. So one of the problems with this algorithm is it ignores word order. In particular, this is a very negative review, "Completely lacking in good taste, good service, and good ambiance". But the word good appears a lot.



This is a lot. Good, good, good. So if you use an algorithm like this that ignores word order and just sums or averages all of the embeddings for the different words, then you end up having a lot of the representation of good in your final feature vector and your classifier will probably think this is a good review even though this is actually very harsh. This is a one-star review.

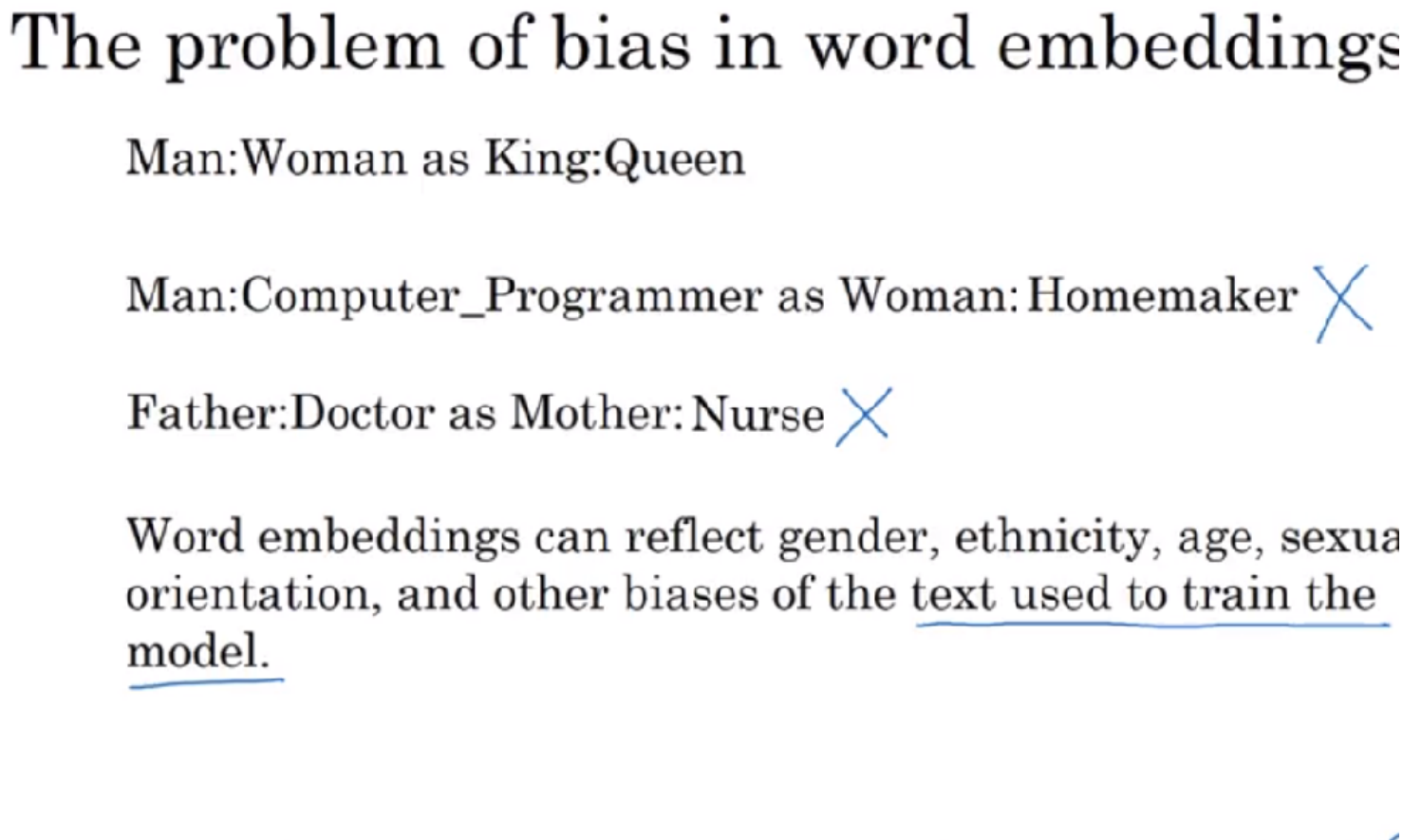
So here's a more sophisticated model which is that, instead of just summing all of your word embeddings, you can instead use a RNN for sentiment classification. So here's what you can do. You can take that review, "Completely lacking in good taste, good service, and good ambiance", and find for each of them, the one-hot vector and so I'm going to just skip the one-hot vector representation but take the one-hot vectors, multiply it by the embedding matrix E as usual, then this gives you the embedding vectors and then you can feed these into an RNN and the job of the RNN is to then compute the representation at the last time step that allows you to predict Y-hat.



So this is an example of a many-to-one RNN architecture which we saw in the previous section and with an algorithm like this, it will be much better at taking word sequence into account and realize that "things are lacking in good taste" is a negative review and "not good" a negative review unlike the previous algorithm, which just sums everything together into a big-word vector and doesn't realize that "not good" has a very different meaning than the words "good" or "lacking in good taste" and so on and so if you train this algorithm, you end up with a pretty decent sentiment classification algorithm and because your word embeddings can be trained from a much larger data set, this will do a better job generalizing to maybe even new words now that you'll see in your training set, such as if someone else says, "Completely absent of good taste, good service, and good ambiance" or something, then even if the word "absent" is not in your label training set, if it was in your 1 billion or 100 billion word corpus used to train the word embeddings, it might still get this right and generalize much better even to words that were in the training set used to train the word embeddings but not necessarily in the label training set that you had for specifically the sentiment classification problem. So that's it for sentiment classification, and I hope this gives you a sense of how once you've learned or downloaded from online a word embedding, this allows you to quite quickly build pretty effective NLP systems.

Debiasing word embeddings

Machine learning and AI algorithms are increasingly trusted to help with, or to make, extremely important decisions and so we like to make sure that as much as possible that they're free of undesirable forms of bias, such as gender bias, ethnicity bias and so on. What I want to do in this section is show you some of the ideas for diminishing or eliminating these forms of bias in word embeddings. When I use the term bias in this section, I don't mean the bias variants. Sense the bias, instead I mean gender, ethnicity, sexual orientation bias. That's a different sense of bias then is typically used in the technical discussion on machine learning. But mostly the problem, we talked about how word embeddings can learn analogies like man is to woman as king is to queen. But what if you ask it, man is to computer programmer as woman is to what? And so the authors of this paper Tolga Bolukbasi, Kai-Wei Chang, James Zou, Venkatesh Saligrama, and Adam Kalai found a somewhat horrifying result where a learned word embedding might output Man:Computer_Programmer as Woman:Homemaker and that just seems wrong and it enforces a very unhealthy gender stereotype. It'd be much more preferable to have algorithm output man is to computer programmer as a woman is to computer programmer and they found also, Father:Doctor as Mother is to what? and the really unfortunate result is that some learned word embeddings would output Mother:Nurse.

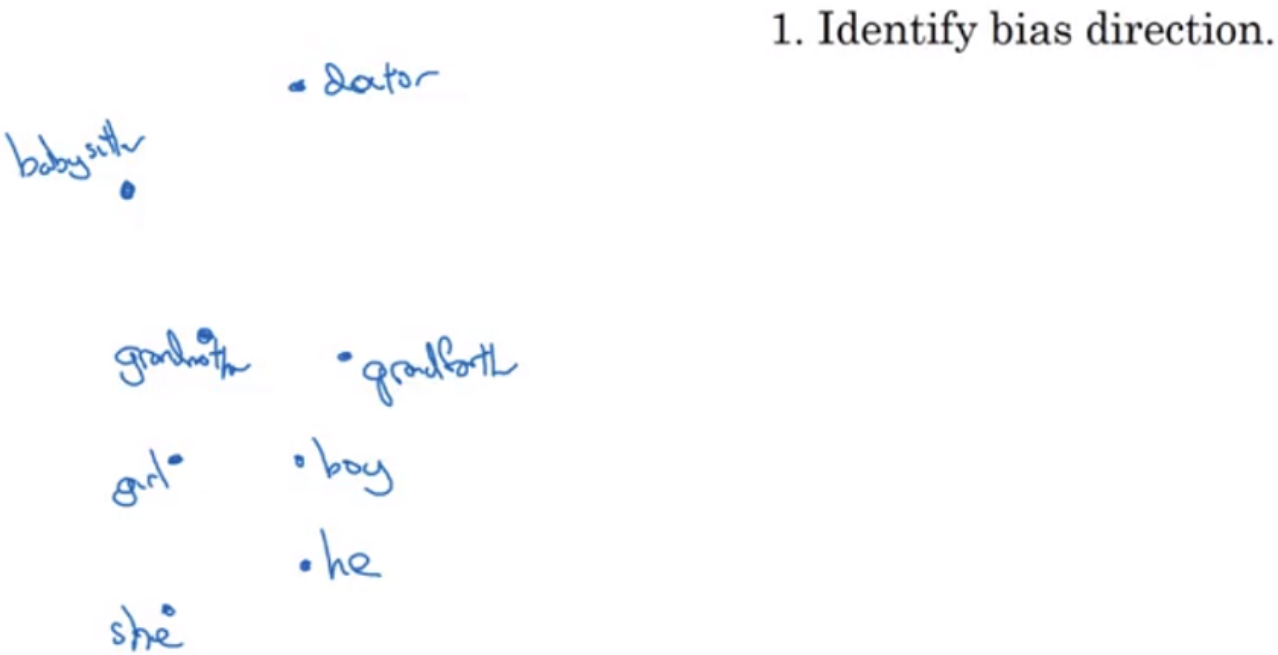


Bolukbasi et. al., 2016. Man is to computer programmer as woman is to homemaker? Debiasing word embeddings]

So word embeddings can reflect the gender, ethnicity, age, sexual orientation, and other biases of the text used to train the model. One that I'm especially passionate about is bias relating to socioeconomic status. I think that every person, whether you come from a wealthy family, or a low income family, or anywhere in between, I think everyone should have great opportunities and because machine learning algorithms are being used to make very important decisions. They're influencing everything ranging from college admissions, to the way people find jobs, to loan applications, whether your application for a loan gets approved, to in the criminal justice system, even sentencing guidelines. Learning algorithms are making very important decisions and so I think it's important that we try to change learning algorithms to diminish as much as is possible, or, ideally, eliminate these types of undesirable biases. Now in the case of word embeddings, they can pick up the biases of the text used to train the model and so the biases they pick up or tend to reflect the biases in text as is written by people. Over many decades, over many centuries, I think humanity has made progress in reducing these types of bias and I think maybe fortunately for AI, I think we actually have better ideas for quickly reducing the bias in AI than for quickly reducing the bias in the human race. Although I think we're by no means done for AI as well and there's still a lot of research and hard work to be done to reduce these types of biases in our learning algorithms. But what I want to do in this video is share with you one example of a set of ideas due to the paper referenced at the bottom by Bolukbasi and others on reducing the bias in word embeddings.

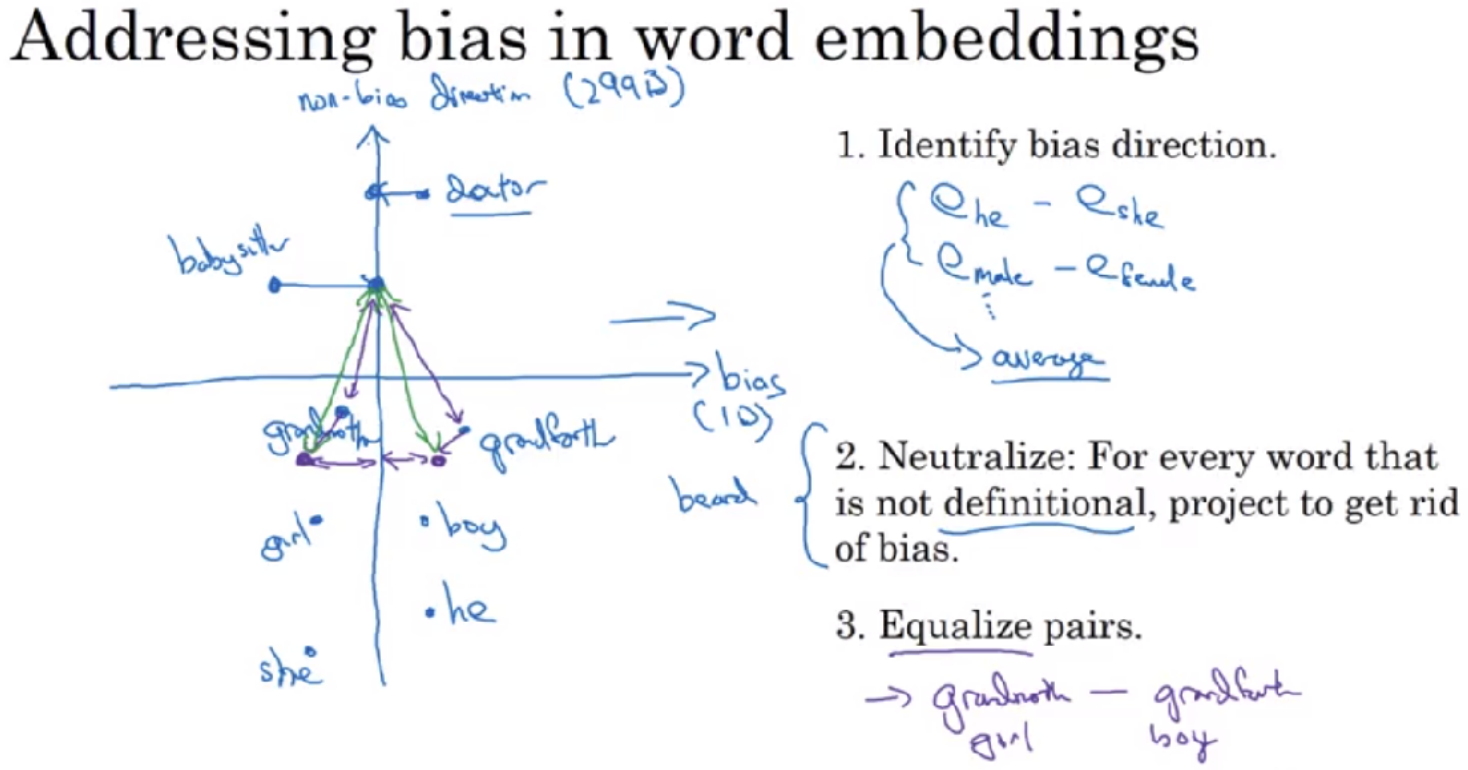
So here's the idea. Let's say that we've already learned a word embedding, so the word babysitter is here, the word doctor is here. We have grandmother here, and grandfather here. Maybe the word girl is embedded there, the word boy is embedded there and maybe she is embedded here, and he is embedded there. check diagram:

Addressing bias in word embeddings



So the first thing we're going to do it is identify the direction corresponding to a particular bias we want to reduce or eliminate and, for illustration, we'll focus on gender bias but these ideas are applicable to all of the other types of bias that we mention on the previous section as well and so how do you identify the direction corresponding to the bias? For the case of gender, what we can do is take the embedding vector for he and subtract the embedding vector for she, because that differs by gender and take e male, subtract e female, and take a few of these and average them, right? And take a few of these differences and basically average them and this will allow you to figure out in this case that what looks like this direction is the gender direction, or the bias direction. Whereas this direction is unrelated to the particular bias we're trying to address. So this is the non-bias direction. And in this case, the bias direction, think of this as a 1D subspace whereas a non-bias direction, this will be 299-dimensional subspace. Okay, and I've simplified the description a little bit in the original paper. The bias direction can be higher than 1-dimensional, and rather than take an average, as I'm describing it here, it's actually found using a more complicated algorithm called a SVU, a **singular value decomposition**. Which is closely related to, if you're familiar with **principle component analysis**, it uses ideas similar to the **PCA algorithm**. After that, the next step is a neutralization step. So for every word that's not definitional, project it to get rid of bias. So there are some words that intrinsically capture gender. So words like grandmother, grandfather, girl, boy, she, he, a gender is intrinsic in the definition. Whereas there are other word like doctor and babysitter that we want to be gender neutral and really, in the more general case, you might want words like doctor or babysitter to be ethnicity neutral or sexual orientation neutral, and so on, but we'll just use gender as the illustrating example here. But so for every word that is not definitional, this basically means not words like grandmother and grandfather, which really have a very legitimate gender component, because, by definition, grandmothers are female, and grandfathers are male. So for words like doctor and babysitter, let's just project them onto this axis to reduce their components, or to eliminate their component, in the bias direction. So reduce their component in this horizontal direction.
So that's the second neutralize step and then the final step is called equalization in which you might have pairs of words such as grandmother and grandfather, or girl and boy, where you want the only difference in their embedding to be the gender and so, why do you want that? Well in this example, the distance, or the similarity, between babysitter and grandmother is actually smaller than the distance between babysitter and grandfather and so this maybe reinforces an unhealthy, or maybe undesirable, bias that grandmothers end up babysitting more than grandfathers. So in the final equalization step, what we'd like to do is to make sure that words like grandmother and grandfather are both exactly the same similarity, or exactly the same distance, from words that should be gender

neutral, such as babysitter or such as doctor. So there are a few linear algebra steps for that. But what it will basically do is move grandmother and grandfather to a pair of points that are equidistant from this axis in the middle and so the effect of that is that now the distance between babysitter, compared to these two words, will be exactly the same and so, in general, there are many pairs of words like this grandmother-grandfather, boy-girl, sorority-fraternity, girlhood-boyhood, sister-brother, niece-nephew, daughter-son, that you might want to carry out through this equalization step.



So the final detail is, how do you decide what word to neutralize? So for example, the word doctor seems like a word you should neutralize to make it non-gender-specific or non-ethnicity-specific. Whereas the words grandmother and grandmother should not be made non-gender-specific and there are also words like beard, right, that it's just a statistical fact that men are much more likely to have beards than women, so maybe beards should be closer to male than female. And so what the authors did is train a classifier to try to figure out what words are definitional, what words should be gender-specific and what words should not be and it turns out that most words in the English language are not definitional, meaning that gender is not part of the definition and it's such a relatively small subset of words like this, grandmother-grandfather, girl-boy, sorority-fraternity, and so on that should not be neutralized and so a linear classifier can tell you what words to pass through the neutralization step to project out this bias direction, to project it on to this essentially 299-dimensional subspace and then, finally, the number of pairs you want to equalize, that's actually also relatively small, and is, at least for the gender example, it is quite feasible to hand-pick most of the pairs you want to equalize. So the full algorithm is a bit more complicated than we have seen here, we can take a look at the paper for the full details.

So to summarize, that reducing or eliminating bias of our learning algorithms is a very important problem because these algorithms are being asked to help with or to make more and more important decisions in society. In this section we have seen just one set of ideas for how to go about trying to address this problem, but this is still a very much an ongoing area of active research by many researchers.

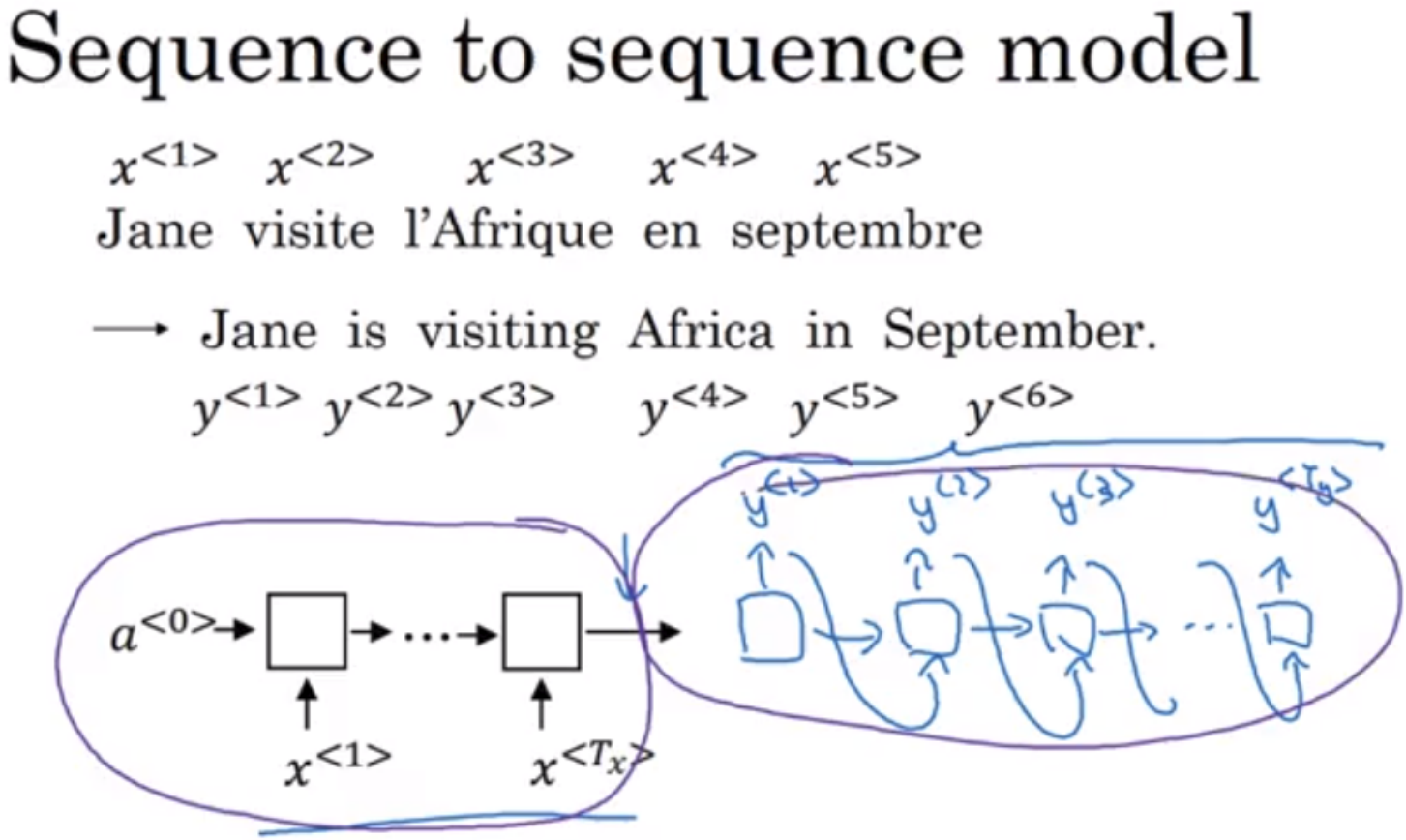
Week 3: Sequence models & Attention mechanism

Sequence models can be augmented using an attention mechanism. This algorithm will help your model understand where it should focus its attention given a sequence of inputs. This week, you will also learn about speech recognition and how to deal with audio data.

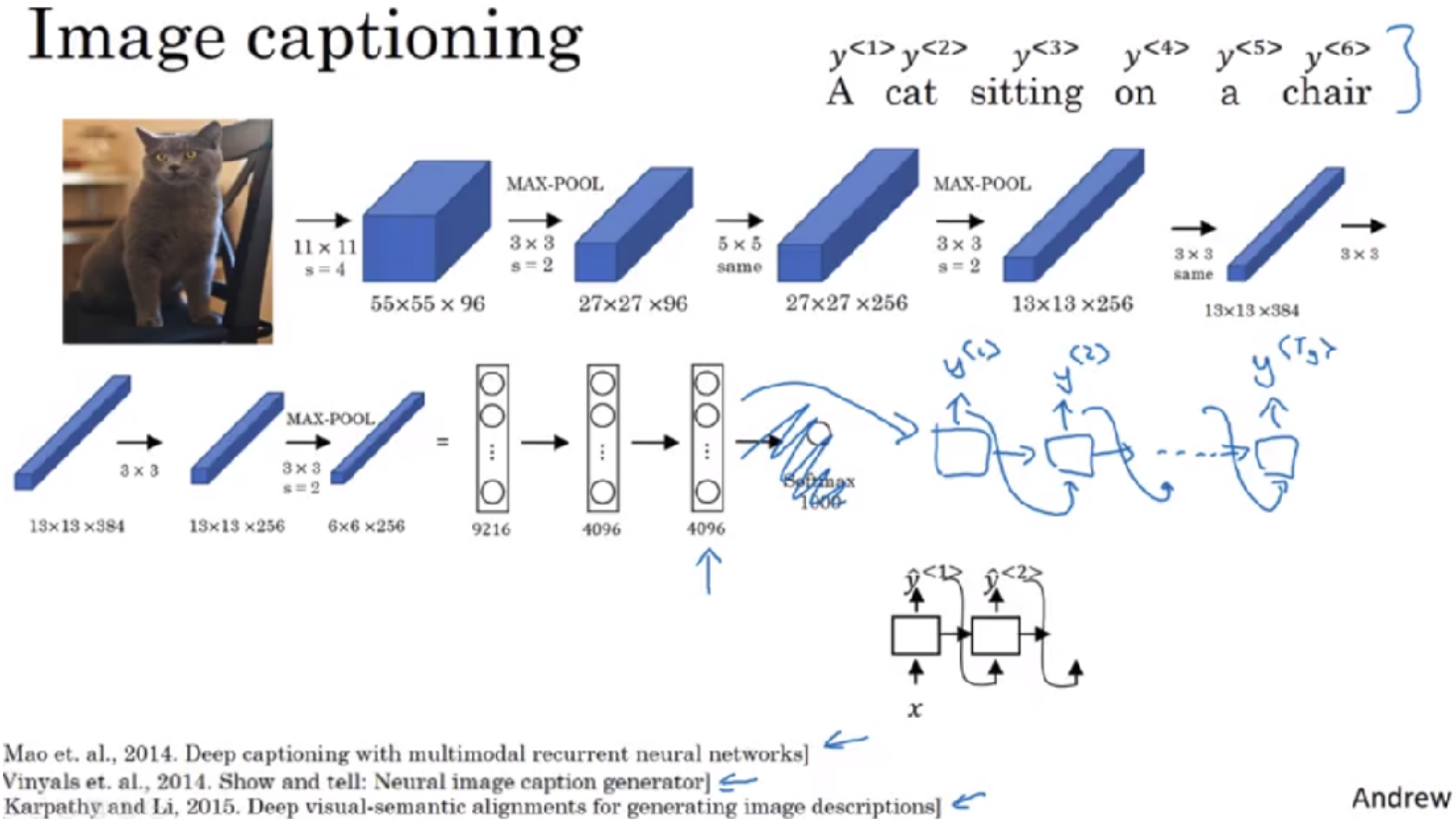
Various sequence to sequence architectures
Object Localization
Basic Models

In this week, you hear about sequence-to-sequence models, which are useful for everything from machine translation to speech recognition. Let's start with the basic models and then later this week you, hear about beam search, the attention model, and we'll wrap up the discussion of models for audio data, like speech. Let's get started.

Let's say you want to input a French sentence like Jane visite l'Afrique en septembre, and you want to translate it to the English sentence, Jane is visiting Africa in September. As usual, let's use $x^{<1>}$ through x, in this case $x^{<5>}$, to represent the words in the input sequence, and we'll use $y^{<1>}$ through $y^{<6>}$ to represent the words in the output sequence. So, how can you train a new network to input the sequence x and output the sequence y? Well, here's something you could do, and the ideas I'm about to present are mainly from these two papers due to Sutskever, Oriol Vinyals, and Quoc Le, and that one by Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwen, and Yoshua Bengio.



First, let's have a network, which we're going to call the encoder network be built as a RNN, and this could be a GRU and LSTM, feed in the input French words one word at a time. And after ingesting the input sequence, the RNN then offers a vector that represents the input sentence. After that, you can build a decoder network which I'm going to draw here, which takes as input the encoding output by the encoder network shown in black on the left, and then can be trained to output the translation one word at a time until eventually it outputs say, the end of sequence or end the sentence token upon which the decoder stops and as usual we could take the generated tokens and feed them to the next [inaudible] in the sequence like we 're doing before when synthesizing text using the language model. One of the most remarkable recent results in deep learning is that this model works, given enough pairs of French and English sentences. If you train the model to input a French sentence and output the corresponding English translation, this will actually work decently well and this model simply uses an encoder network, whose job it is to find an encoding of the input French sentence and then use a decoder network to then generate the corresponding English translation.



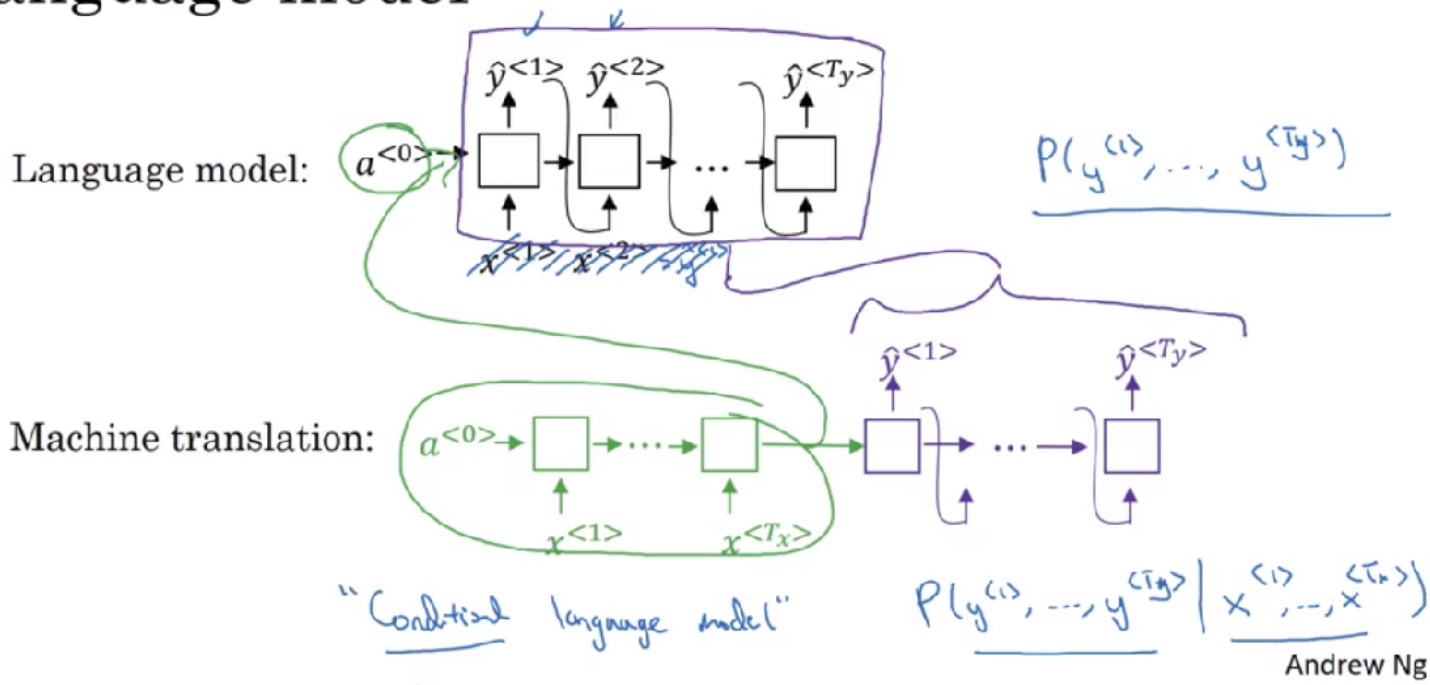
An architecture very similar to this also works for image captioning so given an image like the one shown here, maybe wanted to be captioned automatically as a cat sitting on a chair. So how do you train a new network to input an image and output a caption like that phrase up there? Here's what you can do. From the earlier course on [inaudible] you've seen how you can input an image into a convolutional network, maybe a pre-trained AlexNet, and have that learn an encoding or learn a set of features of the input image. So, this is actually the AlexNet architecture and if we get rid of this final Softmax unit, the pre-trained AlexNet can give you a 4096-dimensional feature vector of which to represent this picture of a cat. And so this pre-trained network can be the encoder network for the image and you now have a 4096-dimensional vector that represents the image. You can then take this and feed it to an RNN, whose job it is to generate the caption one word at a time. So similar to what we saw with machine translation translating from French to English, you can now input a feature vector describing the input and then have it generate an output sequence or output set of words one word at a time and this actually works pretty well for image captioning, especially if the caption you want to generate is not too long. As far as I know, this type of model was first proposed by Junhua Mao, Wei Xu, Yi Yang, Jiang Wang, Zhiheng Huang, and Alan Yuille, although it turns out there were multiple groups coming up with very similar models independently and at about the same time. So two other groups that had done very similar work at about the same time and I think independently of Mao et al were Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan, as well as Andrej Karpathy and Fei-Fei Yi. So, you've now seen how a basic sequence-to-sequence model works, or how a basic image-to-sequence or image captioning model works, but there are some differences between how you would run a model like this, so generating a sequence compared to how you were synthesizing novel text using a language model. One of the key differences is, you don't want a randomly chosen translation, you maybe want the most likely translation, or you don't want a randomly chosen caption, maybe not, but you might want the best caption and most likely caption. So let's see in the next section how you go about generating that.

Picking the most likely sequence

There are some similarities between the sequence to sequence machine translation model and the language models that you have worked within the first week of this course, but there are some significant differences as well. Let's take a look. So, you can think of machine translation as building a conditional language model. Here's what I mean, in language modeling, this was the network we had

built in the first week and this model allows you to estimate the probability of a sentence. That's what a language model does and you can also use this to generate novel sentences, and sometimes when you are writing x1 and x2 here, where in this example, x2 would be equal to y1 or equal to y and one is just a feedback. But x1, x2, and so on were not important. So just to clean this up for this slide, I'm going to just cross these off. X1 could be the vector of all zeros and x2, x3 are just the previous output you are generating. So that was the language model.

Machine translation as building a conditional language model



The machine translation model looks as follows, and I am going to use a couple different colors, green and purple, to denote respectively the coded network in green and the decoded network in purple and you notice that the decoded network looks pretty much identical to the language model that we had up there. So what the machine translation model is, is very similar to the language model, except that instead of always starting along with the vector of all zeros, it instead has an encoded network that figures out some representation for the input sentence, and it takes that input sentence and starts off the decoded network with representation of the input sentence rather than with the representation of all zeros. So, that's why we call this a conditional language model, and instead of modeling the probability of any sentence, it is now modeling the probability of, say, the output English translation, conditions on some input French sentence. So in other words, you're trying to estimate the probability of an English translation. Like, what's the chance that the translation is "**Jane is visiting Africa in September**," but conditions on the input French censors like, "**Jane visite I'Afrique en septembre**." So, this is really the probability of an English sentence conditions on an input French sentence which is why it is a conditional language model. Now, if you want to apply this model to actually translate a sentence from French into English, given this input French sentence, the model might tell you what is the probability of difference in corresponding English translations. So, x is the French sentence, "Jane visite l'Afrique en septembre." And, this now tells you what is the probability of different English translations of that French input. And, what you do not want is to sample outputs at random. If you sample words from this distribution, p of y given x, maybe one time you get a pretty good translation, "Jane is visiting Africa in September." But, maybe another time you get a different translation, "Jane is going to be visiting Africa in September. " Which sounds a little awkward but is not a terrible translation, just not the best one. And sometimes, just by chance, you get, say, others: "In September, Jane will visit Africa." And maybe, just by chance, sometimes you sample a really bad translation: "Her African friend welcomed Jane in September."

Finding the most likely translation

Jane visite l'Afrique en septembre.

$$P(y^{<1>}, \ldots, y^{<T_y>} \mid x)$$

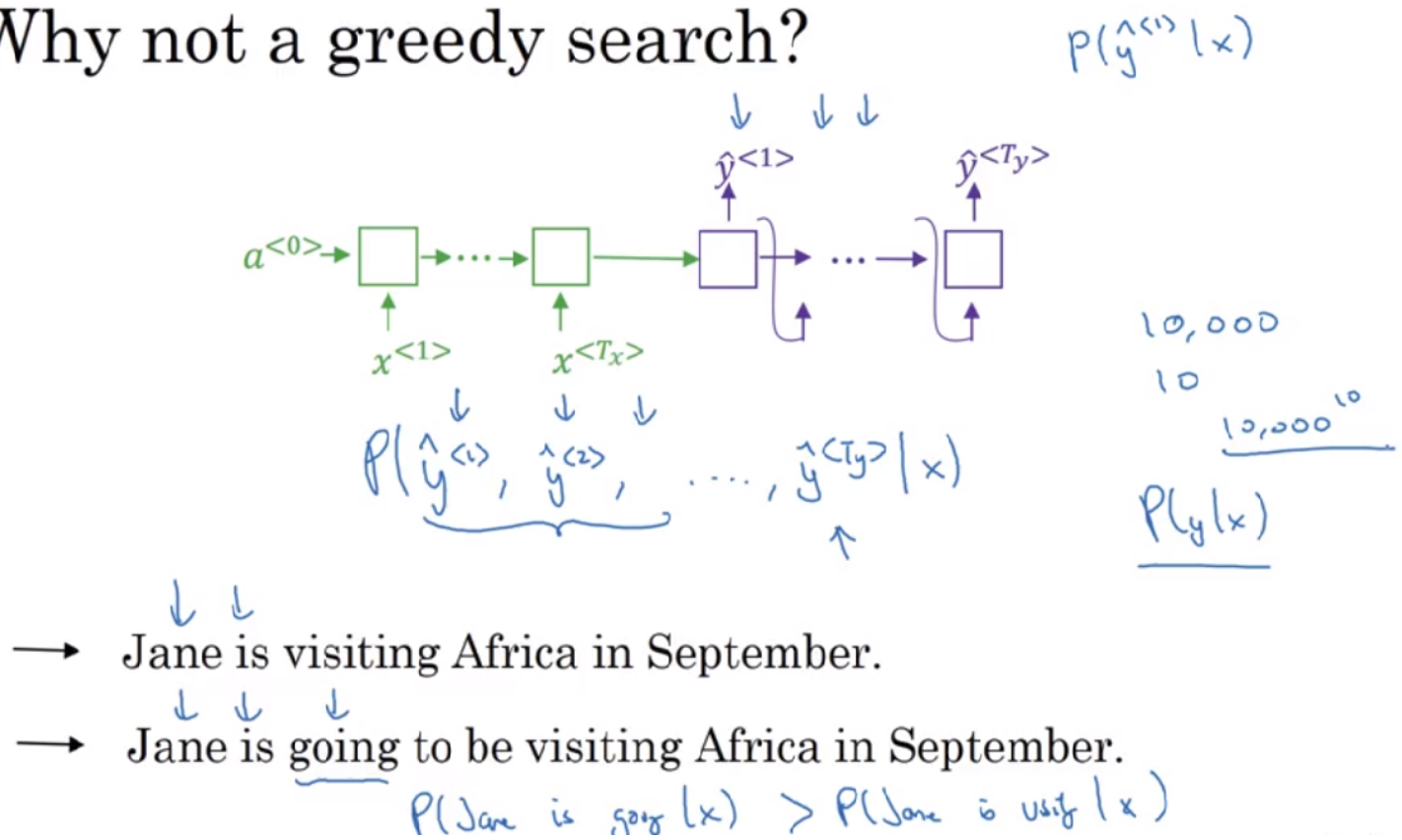
English

French

- → Jane is visiting Africa in September.
- → Jane is going to be visiting Africa in September.
- → In September, Jane will visit Africa.
- → Her African friend welcomed Jane in September.

$$\arg\max_{y^{<1>},\ldots,y^{<T_y>}} P(y^{<1>}, \ldots, y^{<T_y>} \mid x)$$

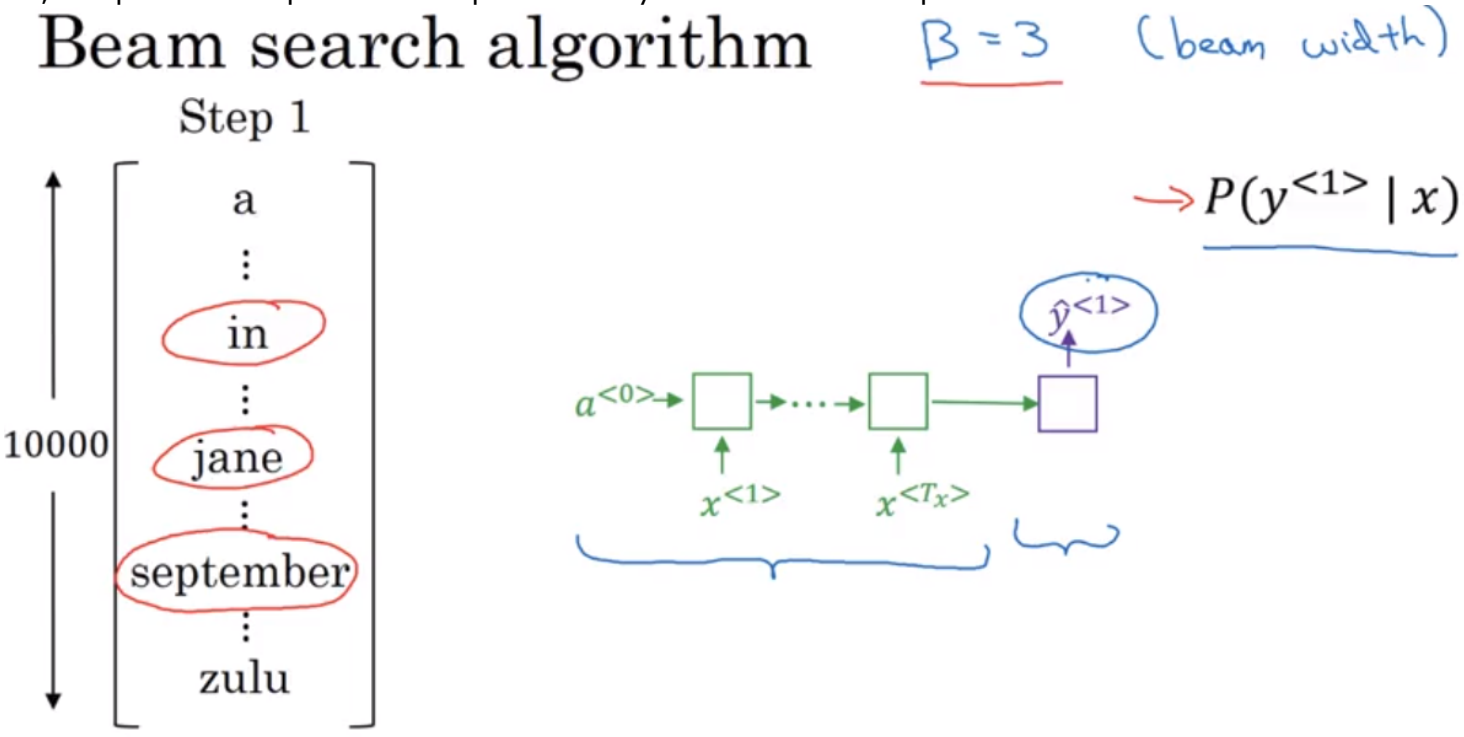
So, when you're using this model for machine translation, you're not trying to sample at random from this distribution. Instead, what you would like is to find the English sentence, y, that maximizes that conditional probability. So in developing a machine translation system, one of the things you need to do is come up with an algorithm that can actually find the value of y that maximizes this term over here. The most common algorithm for doing this is called **beam search**, and it's something you'll see in the next section. But, before moving on to describe beam search, you might wonder, why not just use **greedy search**? So, what is greedy search? Well, greedy search is an algorithm from computer science which says to generate the first word just pick whatever is the most likely first word according to your conditional language model. Going to your machine translation model and then after having picked the first word, you then pick whatever is the second word that seems most likely, then pick the third word that seems most likely. This algorithm is called greedy search.



And, what you would really like is to pick the entire sequence of words, y1, y2, up to yTy, that's there, that maximizes the joint probability of that whole thing and it turns out that the greedy approach, where you just pick the best first word, and then, after having picked the best first word, try to pick the best second word, and then, after that, try to pick the best third word, that approach doesn't really work. To demonstrate that, let's consider the following two translations. The first one is a better translation, so hopefully, in our machine translation model, it will say that p of y given x is higher for the first sentence. It's just a better, more succinct translation of the French input. The second one is not a bad translation, it's just more verbose, it has more unnecessary words. But, if the algorithm has picked "Jane is" as the first two words, because "going" is a more common English word, probably the chance of "Jane is going," given the French input, this might actually be higher than the chance of "Jane is visiting," given the French sentence. So, it's quite possible that if you just pick the third word based on whatever maximizes the probability of just the first three words, you end up choosing option number two. But, this ultimately ends up resulting in a less optimal sentence, in a less good sentence as measured by this model for p of y given x. I know this was may be a slightly hand-wavey argument, but, this is an example of a broader phenomenon, where if you want to find the sequence of words, y1, y2, all the way up to the final word that together maximize the probability, it's not always optimal to just pick one word at a time and, of course, the total number of combinations of words in the English sentence is exponentially larger. So, if you have just 10,000 words in a dictionary and if you're contemplating translations that are up to ten words long, then there are 10000 to the tenth possible sentences that are ten words long. Picking words from the vocabulary size, the dictionary size of 10000 words. So, this is just a huge space of possible sentences, and it's impossible to rate them all, which is why the most common thing to do is use an approximate search out of them and, what an approximate search algorithm does, is it will try, it won't always succeed, but it will to pick the sentence, y, that maximizes that conditional probability and, even though it's not guaranteed to find the value of y that maximizes this, it usually does a good enough job. So, to summarize, in this section, you saw how machine translation can be posed as a conditional language modeling problem. But one major difference between this and the earlier language modeling problems is rather than wanting to generate a sentence at random, you may want to try to find the most likely English sentence, most likely English translation. But the set of all English sentences of a certain length is too large to exhaustively enumerate. So, we have to resort to a search algorithm. So, with that, let's go onto the next section where you'll learn about beam search algorithm.

Beam Search

In this section, we'll learn about the beam search algorithm. In the last section, we remember how for machine translation given an input French sentence, we don't want to output a random English translation, we want to output the best and the most likely English translation. The same is also true for speech recognition where given an input audio clip, you don't want to output a random text transcript of that audio, you want to output the best, maybe the most likely, text transcript. **Beam search is the most widely used algorithm to do this** and in this section, we'll see how to get beam search to work for yourself. Let's just try Beam Search using our running example of the French sentence, "Jane, visite l'Afrique en Septembre". Hopefully being translated into, "Jane, visits Africa in September". The first thing Beam search has to do is try to pick the first words of the English translation, that's going to operate. So here I've listed, say, 10,000 words into vocabulary and to simplify the problem a bit, I'm going to ignore capitalization. So I'm just listing all the words in lower case. So, in the first step of Beam Search, I use this network fragment with the coalition in green and decoalition in purple, to try to evaluate what is the probability of that for a square. So, what's the probability of the first output y, given the input sentence x gives the French input. So, whereas greedy search will pick only the one most likely words and move on, Beam Search instead can consider multiple alternatives. So, the Beam Search algorithm has a parameter called B, which is called the beam width and for this example I'm going to set the beam width to be with the 3 and what this means is Beam search will cause that not just one possibility but consider three at the time. So in particular, let's say evaluating this probability over different choices the first words, it finds that the choices in, Jane and September are the most likely three possibilities for the first words in the English outputs. Then Beam search will store away in computer memory that it wants to try all of three of these words, and if the beam width parameter were said differently, the beam width parameter was 10, then we keep track of not just three but of the ten, most likely possible choices for the first word. So, to be clear in order to perform this first step of Beam search, what you need to do is run the input French sentence through this encoder network and then this first step will then decode the network, this is a softmax output overall 10,000 possibilities. Then you would take those 10,000 possible outputs and keep in memory which were the top three.



Let's go into the second step of Beam search. Having picked in, Jane and September as the three most likely choice of the first word, what Beam search will do now, is for each of these three choices consider what should be the second word, so after "in" maybe a second word is "a" or maybe as Aaron, I'm just listing words from the vocabulary, from the dictionary or somewhere down the list will be September, somewhere down the list there's visit and then all the way to z and then the last word is zulu. So, to evaluate the probability of second word, it will use this new network fragments where

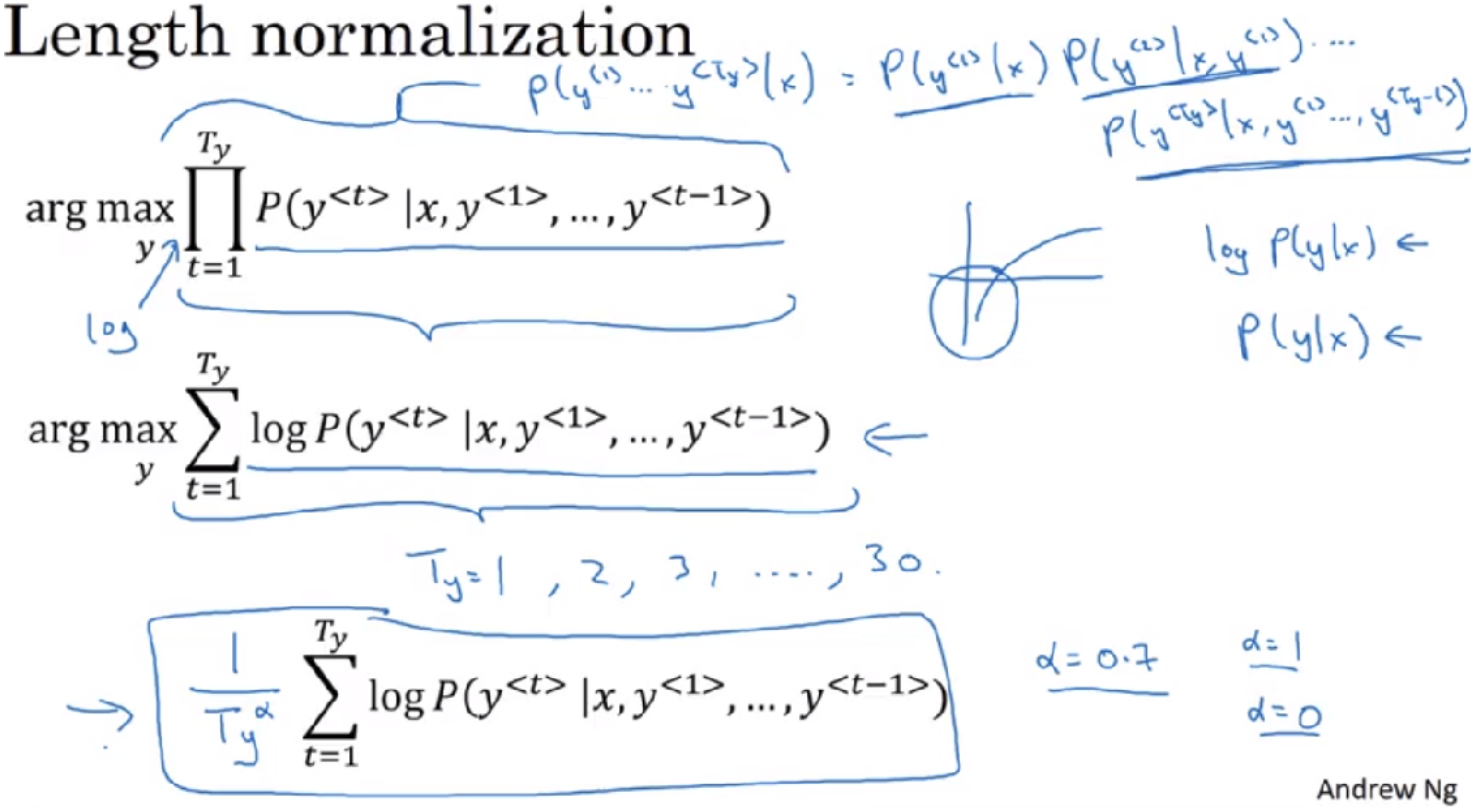
is coder in green and for the decoder portion when trying to decide what comes after in. Remember the decoder first outputs, yhat$^{<1>}$. So, I'm going to set to this y hat one to the word "in" as it goes back in. So there's the word **"in"** because it decided for now. That's because It trying to figure out that the first word was "in", what is the second word, and then this will output I guess yhat$^{<2>}$ and so by hard wiring y hat one here, really the inputs here to be the first words "in" this time were fragment can be used to evaluate whether it's the probability of the second word given the input french sentence and that the first words of the translation has been the word "in". Now notice that what we also need help out in this second step would be assertions to find the pair of the first and second words that is most likely it's not just a second where is most likely that the pair of the first and second whereas the most likely and by the rules of conditional probability. This can be expressed as P of the first words times P of probability of the second words. Which you are getting from this network fragment and so if for each of the three words you've chosen "in", "Jane," and "September" you save away this probability then you can multiply them by this second probabilities to get the probability of the first and second words. So now you've seen how if the first word was "in" how you can evaluate the probability of the second word. Now at first it was "Jane" you do the same thing. The sentence could be "Jane a"," Jane Aaron", and so on down to "Jane is", "Jane visits" and so on. And you will use this in your network fragments let me draw this in as well where here you will hardwire, yhat$^{<1>}$ to be Jane. And so with the First word yhat$^{<1>}$'s hard wired as Jane than just the network fragments can tell you what's the probability of the second words to me. And given that the first word is "Jane". And then same as above you can multiply with P(y$^{<1>}$) to get the probability of Y1 and Y2 for each of these 10,000 different possible choices for the second word. And then finally do the same thing for September although words from a down to Zulu and use this network fragment. That just goes in as well to see if the first word was September. What was the most likely options for the second words. So for this second step of beam search because we're continuing to use a beam width of three and because there are 10,000 words in the vocabulary you'd end up considering three times 10000 or thirty thousand possibilities because there are 10,000 here, 10,000 here, 10,000 here as the beam width times the number of words in the vocabulary and what you do is you evaluate all of these 30000 options according to the probably the first and second words and then pick the top three. So with a cut down, these **30,000 possibilities down to three again down the beam** width rounded again so let's say that 30,000 choices, the most likely were in September and say Jane is, and Jane visits sorry this bit messy but those are the most likely three out of the 30,000 choices then that's what Beam's search would memorize away and take on to the next step being surge. So notice one thing if beam search decides that the most likely choices are the first and second words are in September, or Jane is, or Jane visits. Then what that means is that it is now rejecting September as a candidate for the first words of the output English translation so we're now down to two possibilities for the first words but we still have a beam width of three keeping track of three choices for pairs of y$^{<1>}$, y$^{<2>}$before going onto the third step of beam search. Just want to notice that because of beam width is equal to three, every step you instantiate three copies of the network to evaluate these partial sentence fragments and the output and it's because of beam width is equal to three that you have three copies of the network with different choices for the first words, but these three copies of the network can be very efficiently used to evaluate all 30,000 options for the second word. So just don't instantiate 30,000 copies of the network or three copies of the network to very quickly evaluate all 10,000 possible outputs at that softmax output say for Y2. Let's just quickly illustrate one more step of beam search. So said that the most likely choices for first two words were in September, Jane is, and Jane visits and for each of these pairs of words which we should have saved the way in computer memory the probability of y$^{<1>}$, y$^{<2>}$ given the input X given the French sentence X. So similar to before, we now want to consider what is the third word. So in September a? In September Aaron? All the way down to is in September Zulu and to evaluate possible choices for the third word, you use this network fragments where you Hardwire the first word here to be in the second word to be September and so this network fragment allows you to evaluate what's the probability of the third word given the input French sentence X and given that the first two words are in September and English output and then you do the same thing for the second fragment. So like so and same thing for Jane visits and so beam search will then once again pick the top three possibilities may be that things in September. Jane is a likely outcome or Jane is visiting is likely or maybe Jane visits Africa is likely for that first three words and then it keeps going and then you go onto the fourth step of beam

search hat one more word and on it goes and the outcome of this process hopefully will be that adding one word at a time that Beam search will decide that. Jane visits Africa in September will be terminated by the end of sentence symbol using that system is quite common. They'll find that this is a likely output English sentence and you'll see more details of this yourself. So with a beam of three being searched considers three possibilities at a time. Notice **that if the beam width was said to be equal to one, say cause there's only one, then this essentially becomes the greedy search algorithm which we had discussed in the last section but by considering multiple possibilities say three or ten or some other number at the same time beam search will usually find a much better output sentence than greedy search**. You've now seen how Beam Search works but it turns out there's some additional tips and tricks from our partners that help you to make beam search work even better. Let's go onto the next section to take a look.

Refinements to Beam Search

In the last section, you saw the basic beam search algorithm. In this section, we'll learn some little changes that make it work even better. Length normalization is a small change to the beam search algorithm that can help you get much better results. Here's what it is. Beam search is maximizing this probability and this product here is just expressing the observation that $P(y^{<1>})$ up to $P(y^{<Ty>})$, given x, can be expressed as $P(y^{<1>})$ given x times $P(y^{<2>})$, given x and y1 times dot dot dot, up to I guess p of y Ty given x and y1 up to y t1-1. Maybe this notation is a bit more scary and more intimidating than it needs to be, but this is that probabilities that you see previously. Now, if you're implementing these, these probabilities are all numbers less than 1. Often they're much less than 1 and multiplying a lot of numbers less than 1 will result in a tiny, tiny, tiny number, which can result in numerical underflow. Meaning that it's too small for the floating part representation in your computer to store accurately. So in practice, instead of maximizing this product, we will take logs and if you insert a log there, then log of a product becomes a sum of a log, and maximizing this sum of log probabilities should give you the same results in terms of selecting the most likely sentence y. So by taking logs, you end up with a more numerically stable algorithm that is less prone to rounding errors, numerical rounding errors, or to really numerical underflow and because the log function, that's the logarithmic function, this is strictly monotonically increasing function, maximizing P(y) and because the logarithmic function, here's the log function, is a strictly monotonically increasing function, we know that maximizing log P(y) given x should give you the same result as maximizing P(y) given x. As in the same value of y that maximizes this should also maximize that. So in most implementations, you keep track of the sum of logs of the probabilities rather than the protocol of probabilities. Now, there's one other change to this objective function that makes the machine translation algorithm work even better. Which is that, if you referred to this original objective up here, if you have a very long sentence, the probability of that sentence is going to be low, because you're multiplying as many terms here. Lots of numbers are less than 1 to estimate the probability of that sentence. And so if you multiply all the numbers that are less than 1 together, you just tend to end up with a smaller probability and so this objective function has an undesirable effect, that maybe it unnaturally tends to prefer very short translations. It tends to prefer very short outputs. Because the probability of a short sentence is determined just by multiplying fewer of these numbers are less than 1 and so the product would just be not quite as small and by the way, the same thing is true for this. The log of our probability is always less than or equal to 1. You're actually in this range of the log. So the more terms you have together, the more negative this thing becomes. So there's one other change to the algorithm that makes it work better, which is instead of using this as the objective you're trying to maximize, one thing you could do is normalize this by the number of words in your translation. And so this takes the average of the log of the probability of each word. And this significantly reduces the penalty for outputting longer translations. And in practice, as a heuristic instead of dividing by Ty, by the number of words in the output sentence, sometimes you use a softer approach. We have Ty to the power of alpha, where maybe alpha is equal to 0.7. So if alpha was equal to 1, then yeah, completely normalizing by length. If alpha was equal to 0, then, well, Ty to the 0 would be 1, then you're just not normalizing at all. And this is somewhat in between full normalization, and no normalization, and alpha's another hyper parameter you have within that you can tune to try to get the best results and have to admit, using alpha this way, this is a heuristic or this is a hack. There isn't a great theoretical justification for it, but people have found this works well.

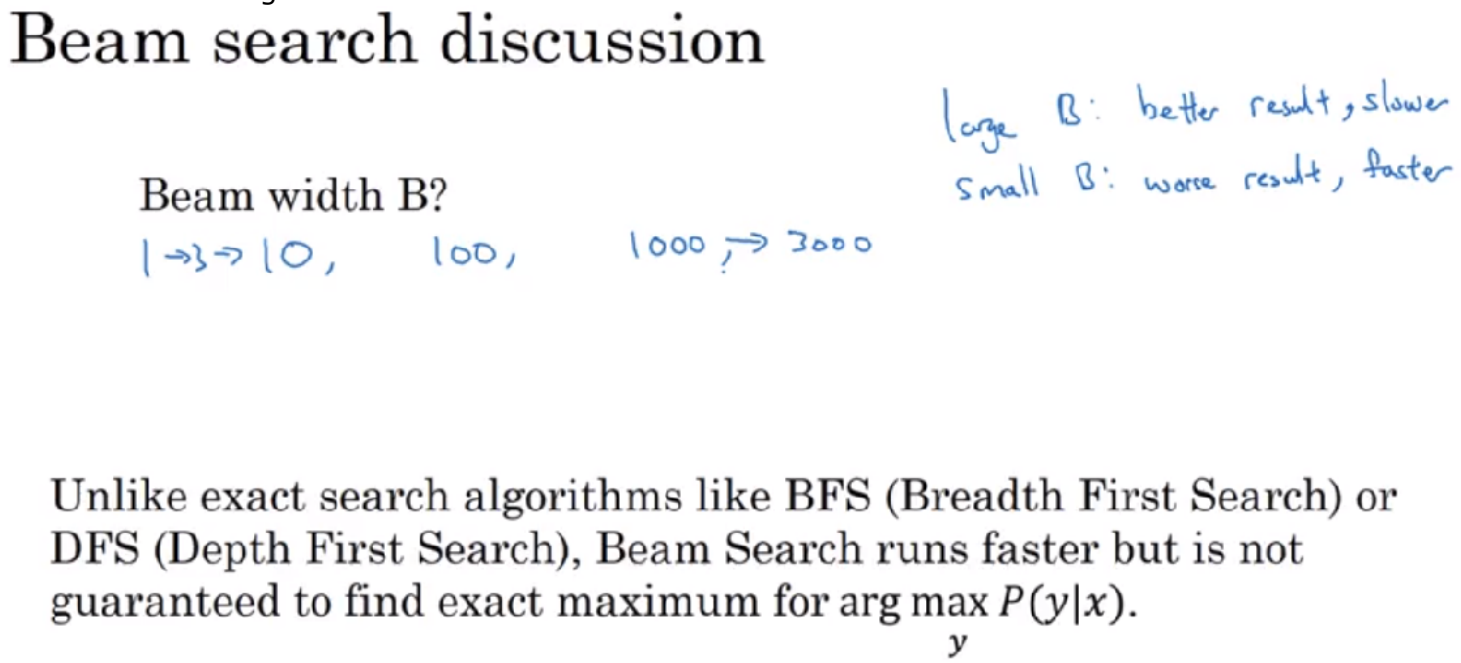
People have found that it works well in practice, so many groups will do this. And you can try out different values of alpha and see which one gives you the best result.



So just to wrap up how you run beam search, as you run beam search you see a lot of sentences with length equal 1, a lot of sentences with length equal 2, a lot of sentences with length equals 3 and so on, and maybe you run beam search for 30 steps and you consider output sentences up to length 30, let's say. And so with beam with a 3, you will be keeping track of the top three possibilities for each of these possible sentence lengths, 1, 2, 3, 4 and so on, up to 30. Then, you would look at all of the output sentences and score them against this score and so you can take your top sentences and just compute this objective function onto sentences that you have seen through the beam search process. And then finally, of all of these sentences that you validate this way, you pick the one that achieves the highest value on this normalized log probability objective. Sometimes it's called a normalized log likelihood objective and then that would be the final translation, your outputs.

Finally, a few implementational details, how do you choose the beam width B? The larger B is, the more possibilities you're considering, and does the better the sentence you probably find. But the larger B is, the more computationally expensive your algorithm is, because you're also keeping a lot more possibilities around. All right, so finally, let's just wrap up with some thoughts on how to choose the beam width B. So here are the pros and cons of setting B to be very large versus very small. If the beam width is very large, then you consider a lot of possibilities, and so you tend to get a better result because you are consuming a lot of different options, but it will be slower. And the memory requirements will also grow, will also be compositionally slower. Whereas if you use a very small beam width, then you get a worse result because you're just keeping less possibilities in mind as the algorithm is running. But you get a result faster and the memory requirements will also be lower. So in the previous section, we used in our running example a beam width of three, so we're keeping three possibilities in mind. In practice, that is on the small side. In production systems, it's not uncommon to see a beam width maybe around 10, and I think beam width of 100 would be considered very large for a production system, depending on the application. But for research systems where people want to squeeze out every last drop of performance in order to publish the paper with the best possible result. It's not uncommon to see people use beam widths of 1,000 or 3,000, but this is very application, that's why it's a domain dependent. So I would say try other variety of values of B as you work through your application. **But when B gets very large, there is often diminishing returns**. So for many applications, I would expect to see a huge gain as you go from a beam width of 1, which is very greedy search, to 3, to maybe 10. But the gains as you go from 1,000 to 3,000 in beam width might not be as big and for those of you that have taken maybe a

lot of computer science courses before, if you're familiar with computer science search algorithms like **BFS, Breadth First Search, or DFS, Depth First Search**. The way to think about beam search is that, unlike those other algorithms which you have learned about in a computer science algorithms course, and don't worry about it if you've not heard of these algorithms. But if you've heard of Breadth First Search and Depth First Search then unlike those algorithms, which are exact search algorithms. Beam search runs much faster but does not guarantee to find the exact maximum for this arg max that you would like to find. If you haven't heard of breadth first search or depth first search, don't worry about it, it's not important for our purposes. But if you have, this is how beam search relates to those algorithms.



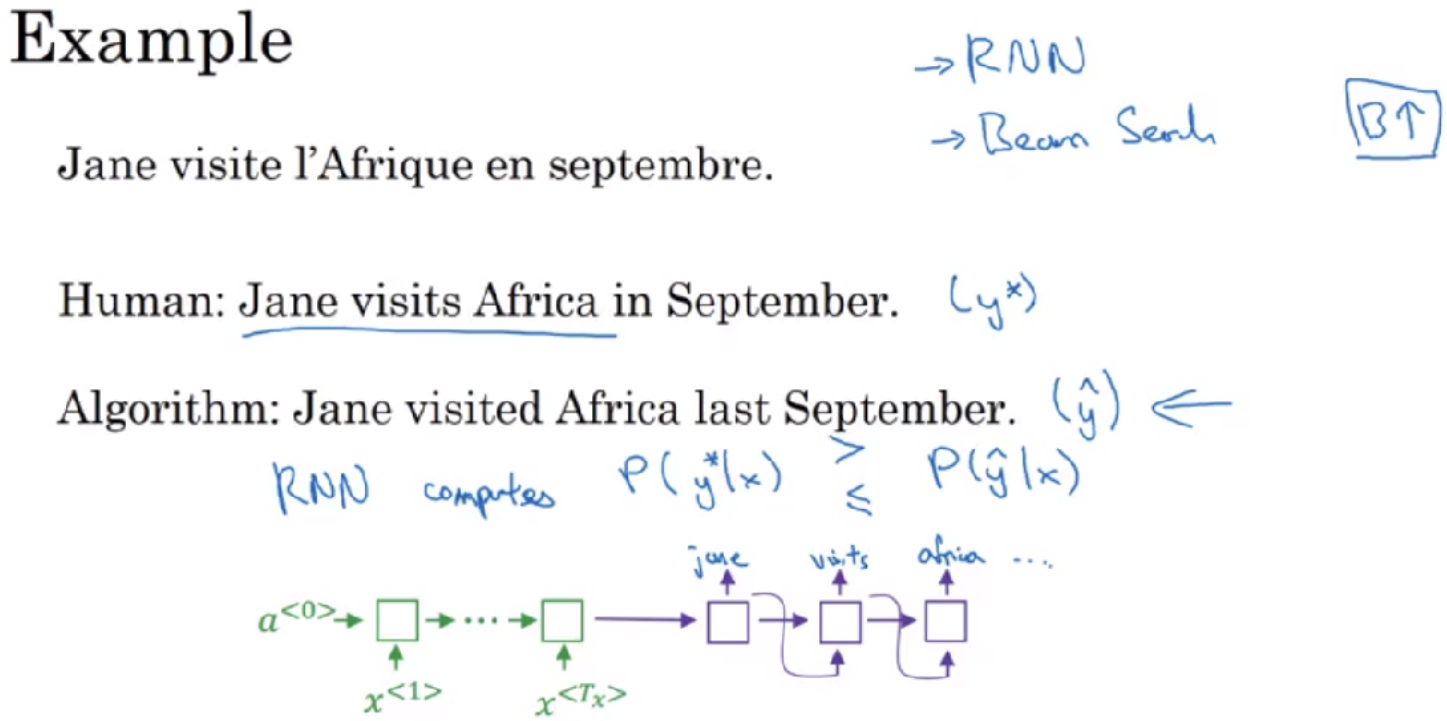
So that's it for beam search, which is a widely used algorithm in many production systems, or in many commercial systems. Now, in the circles in the sequence of courses of deep learning, we talked a lot about error analysis. It turns out, one of the most useful tools I've found is to be able to do error analysis on beam search. So you sometimes wonder, should I increase my beam width? Is my beam width working well enough? And there's some simple things you can compute to give you guidance on whether you need to work on improving your search algorithm. Let's talk about that in the next section.

Error analysis in beam search

In the third course of this sequence of five courses, we saw how error analysis can help you focus your time on doing the most useful work for your project. Now, beam search is an approximate search algorithm, also called a heuristic search algorithm and so it doesn't always output the most likely sentence. It's only keeping track of B equals 3 or 10 or 100 top possibilities. So what if beam search makes a mistake? In this section, we'll learn how error analysis interacts with beam search and how we can figure out whether it is the beam search algorithm that's causing problems and worth spending time on or whether it might be your RNN model that is causing problems and worth spending time on. Let's take a look at how to do error analysis with beam search.

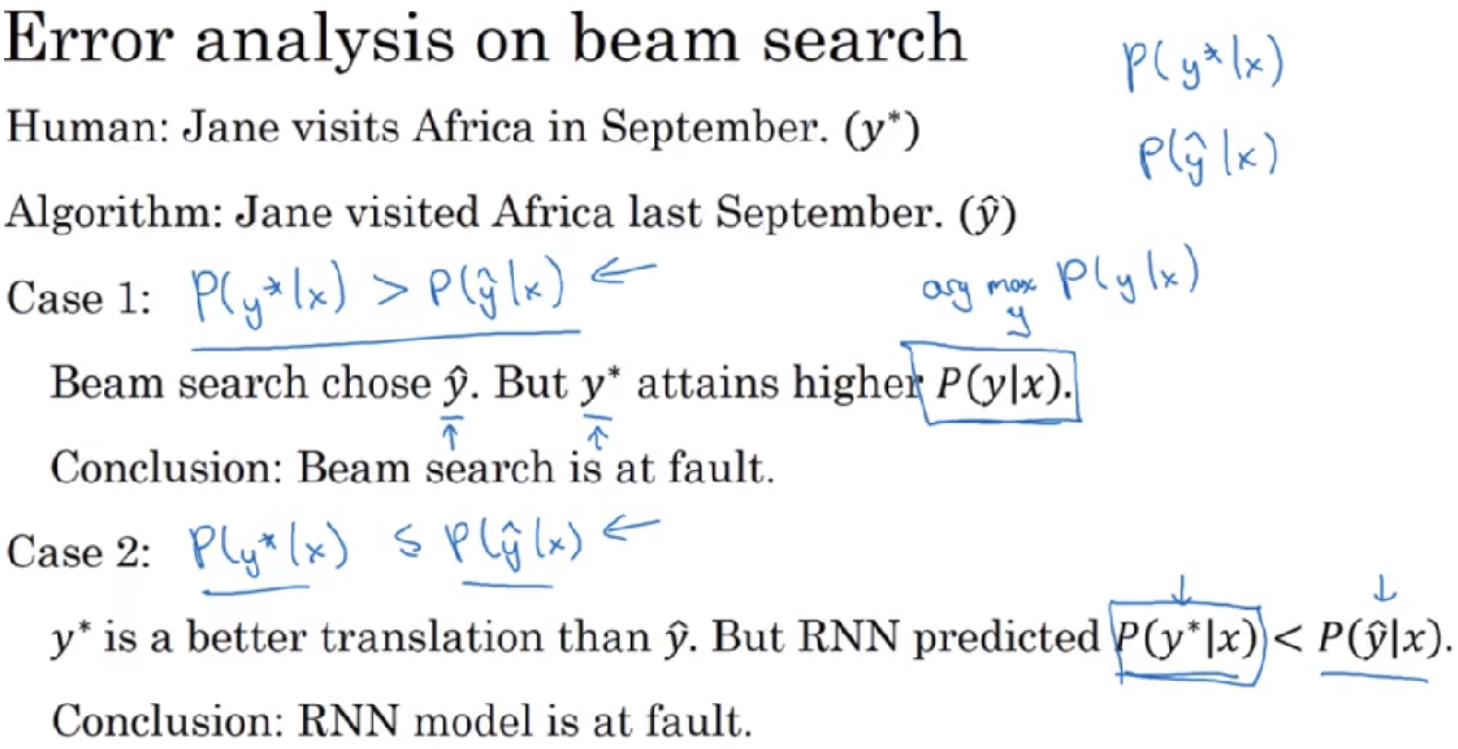
Let's use this example of Jane visite l'Afrique en septembre. So let's say that in your machine translation dev set, your development set, the human provided this translation and Jane visits Africa in September, and I'm going to call this y*. So it is a pretty good translation written by a human. Then let's say that when you run beam search on your learned RNN model and your learned translation model, it ends up with this translation, which we will call yhat, Jane visited Africa last September, which is a much worse translation of the French sentence. It actually changes the meaning, so it's not a good translation. Now, your model has two main components. There is a neural network model, the sequence to sequence model. We shall just call this your RNN model. It's really an encoder and a decoder and you have your beam search algorithm, which you're running with some beam width b. And wouldn't it be nice if you could attribute this error, this not very good translation, to one of these two components? Was it the RNN or really the neural network that is more to blame,

or is it the beam search algorithm, that is more to blame? And what you saw in the third course of the sequence is that it's always tempting to collect more training data that never hurts. So in similar way, it's always tempting to increase the beam width that never hurts or pretty much never hurts. But just as getting more training data by itself might not get you to the level of performance you want. In the same way, increasing the beam width by itself might not get you to where you want to go. But how do you decide whether or not improving the search algorithm is a good use of your time? So just how you can break the problem down and figure out what's actually a good use of your time. Now, the RNN, the neural network, what was called RNN really means the encoder and the decoder. It computes P(y given x). So for example, for a sentence, Jane visits Africa in September, you plug in Jane visits Africa. Again, I'm ignoring upper versus lowercase now, right, and so on and this computes P(y given x). So it turns out that the most useful thing for you to do at this point is to compute using this model to compute P(y* given x) as well as to compute P(yhat given x) using your RNN model and then to see which of these two is bigger. So it's possible that the left side is bigger than the right hand side. It's also possible that P(y*) is less than P(yhat) actually, or less than or equal to, right? Depending on which of these two cases hold true, you'd be able to more clearly describe this particular error, this particular bad translation to one of the RNN or the beam search algorithm being had greater fault. So let's take out the logic behind this.

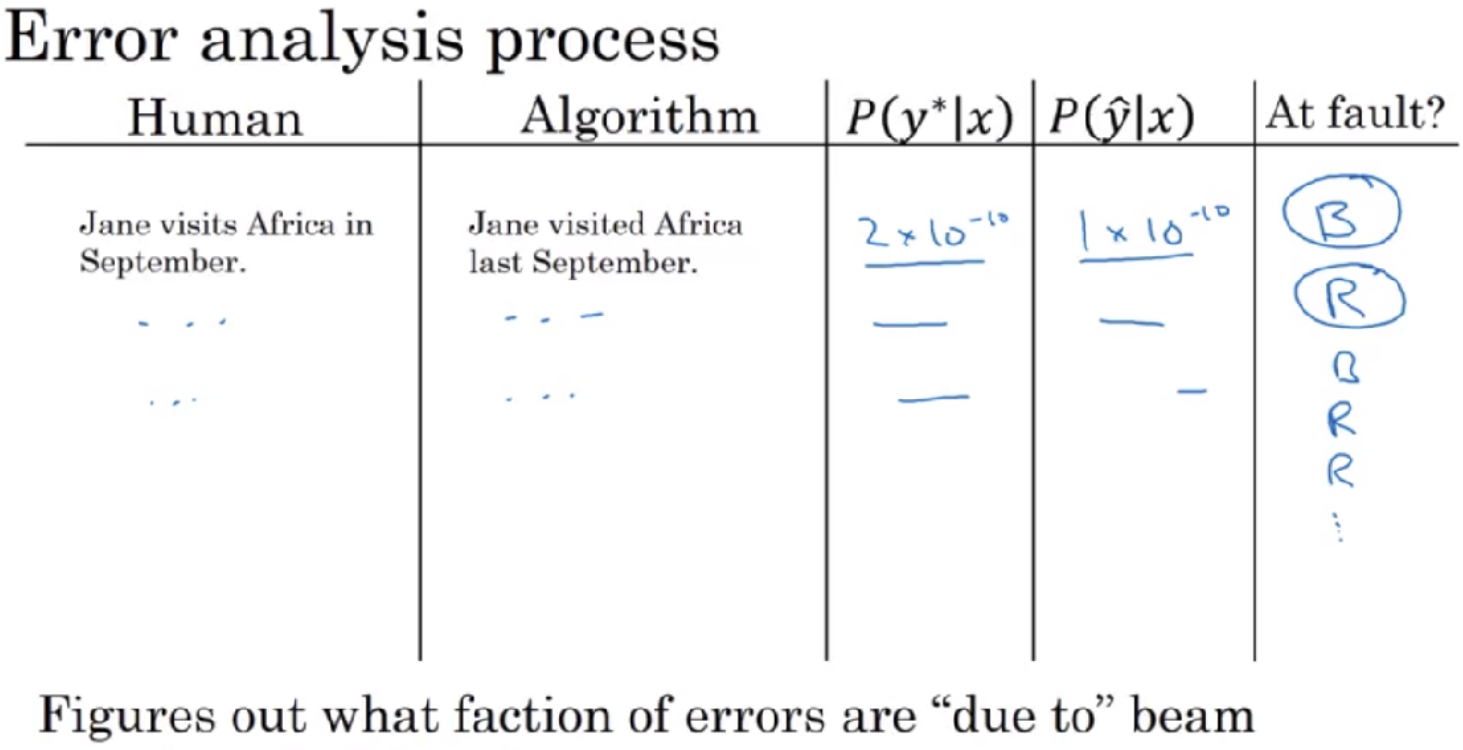


Here are the two sentences from the previous slide. And remember, we're going to compute P(y* given x) and P(y-hat given x) and see which of these two is bigger. So there are going to be two cases. In case 1, P(y* given x) as output by the RNN model is greater than P(y-hat given x). What does this mean? Well, the beam search algorithm chose yhat, right? The way you got y-hat was you had an RNN that was computing P(y given x) and beam search's job was to try to find a value of y that gives that arg max. But in this case, y* actually attains a higher value for P(y given x) than the y-hat. So what this allows you to conclude is beam search is failing to actually give you the value of y that maximizes P(y given x) because the one job that beam search had was to find the value of y that makes this really big. But it choose yhat, the y* actually gets a much bigger value. So in this case, you could conclude that beam search is at fault. Now, how about the other case? In case 2, P(y* given x) is less than or equal to P(y-hat given x), right? And then either this or this has gotta be true. So either case 1 or case 2 has to hold true. What do you conclude under case 2? Well, in our example, y* is a better translation than y-hat. But according to the RNN, P(y*) is less than P(yhat), so saying that y* is a less likely output than yhat. So in this case, it seems that the RNN model is at fault and it might be worth spending more time working on the RNN. There's some subtleties pertaining to length normalizations that I'm glossing over. And if you are using some sort of length normalization, instead of evaluating these probabilities, you should be evaluating the optimization objective that takes into account length normalization. But ignoring that complication for now, in this

case, what this tells you is that even though y* is a better translation, the RNN ascribed y* in lower probability than the inferior translation. So in this case, I will say the RNN model is at fault. So the error analysis process looks as follows. You go through the development set and find the mistakes that the algorithm made in the development set and so in this example, let's say that P(y* given x) was 2 x 10 to the -10, whereas, P(yhat given x) was 1 x 10 to the -10. Using the logic from the previous slide, in this case, we see that beam search actually chose y-hat, which has a lower probability than y*.



So I will say beam search is at fault. So I'll abbreviate that B. And then you go through a second mistake or second bad output by the algorithm, look at these probabilities and maybe for the second example, you think the model is at fault. I'm going to abbreviate the RNN model with R. And you go through more examples and sometimes the beam search is at fault, sometimes the model is at fault, and so on and through this process, you can then carry out error analysis to figure out what fraction of errors are due to beam search versus the RNN model. And with an error analysis process like this, for every example in your dev sets, where the algorithm gives a much worse output than the human translation, you can try to describe the error to either the search algorithm or to the objective function, or to the RNN model that generates the objective function that beam search is supposed to be maximizing. And through this, you can try to figure out which of these two components is responsible for more errors. And only if you find that beam search is responsible for a lot of errors, then maybe is we're working hard to increase the beam width. Whereas in contrast, if you find that the RNN model is at fault, then you could do a deeper layer of analysis to try to figure out if you want to add regularization, or get more training data, or try a different network architecture, or something else and so a lot of the techniques that you saw in the third course in the sequence will be applicable there. So that's it for error analysis using beam search. I found this particular error analysis process very useful whenever you have an approximate optimization algorithm, such as beam search that is working to optimize some sort of objective, some sort of cost function that is output by a learning algorithm, such as a sequence-to-sequence model or a sequence-to-sequence RNN that we've been discussing in these lectures. So with that, I hope that you'll be more efficient at making these types of models work well for your applications.



Bleu Score

One of the challenges of machine translation is that, given a French sentence, there could be multiple English translations that are equally good translations of that French sentence. So how do you evaluate a machine translation system if there are multiple equally good answers, unlike, say, image recognition where there's one right answer? You just measure accuracy. If there are multiple great answers, how do you measure accuracy? The way this is done conventionally is through something called the BLEU score. Let's say you are given a French sentence Le chat est sur le tapis. And you are given a reference, human generated translation of this, which is the the cat is on the mat. But there are multiple, pretty good translations of this. So a different human, different person might translate it as there is a cat on the mat and both of these are actually just perfectly fine translations of the French sentence. What the BLEU score does is given a machine generated translation, it allows you to automatically compute a score that measures how good is that machine translation and the intuition is so long as the machine generated translation is pretty close to any of the references provided by humans, then it will get a high BLEU score. BLEU, stands for bilingual evaluation, Understudy. So in the theater world, an understudy is someone that learns the role of a more senior actor so they can take over the role of the more senior actor, if necessary and motivation for BLEU is that, whereas you could ask human evaluators to evaluate the machine translation system, the BLEU score is an understudy, could be a substitute for having humans evaluate every output of a machine translation system. So the BLEU score was due to Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. This paper has been incredibly influential, and is, actually, quite a readable paper. So I encourage you to take a look if you have time. So, the intuition behind the BLEU score is we're going to look at the machine generated output and see if the types of words it generates appear in at least one of the human generated references and so these human generated references would be provided as part of the depth set or as part of the test set.

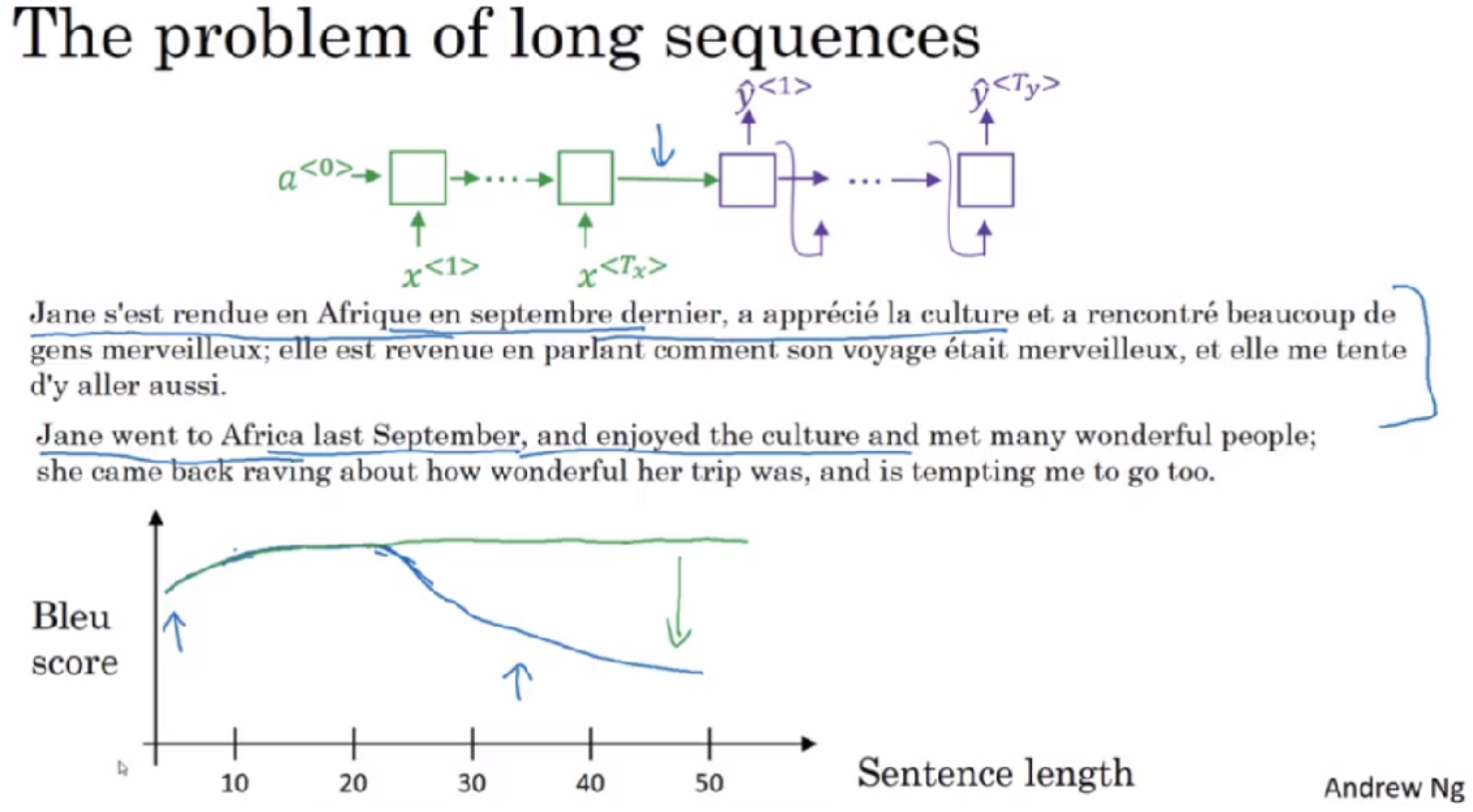
There is a great importance of having a single real number evaluation metric. Because it allows you to try out two ideas, see which one achieves a higher score, and then try to stick with the one that achieved the higher score. So the reason the BLEU score was revolutionary for machine translation was because this gave a pretty good, by no means perfect, but pretty good single real number evaluation metric and so that accelerated the progress of the entire field of machine translation. This section gave us a sense of how the BLEU score works. In practice, few people would implement a BLEU score from scratch. There are open source implementations that you can download and just use to evaluate your own system. But today, BLEU score is used to evaluate many systems that generate text, such as machine translation systems, as well as the example I showed briefly earlier of image captioning systems where you would have a system, have a neural network

generated image caption and then use the BLEU score to see how much that overlaps with maybe a reference caption or multiple reference captions that were generated by people. So the BLEU score is a useful single real number evaluation metric to use whenever you want your algorithm to generate a piece of text and you want to see whether it has similar meaning as a reference piece of text generated by humans. This is not used for speech recognition, because in speech recognition, there's usually one ground truth and you just use other measures to see if you got the speech transcription on pretty much, exactly word for word correct. But for things like image captioning, and multiple captions for a picture, it could be about equally good or for machine translations. There are multiple translations, but equally good. The BLEU score gives you a way to evaluate that automatically and therefore speed up your development.

Attention Model Intuition

For most of this week, we've been using a Encoder-Decoder architecture for machine translation. Where one R and N reads in a sentence and then different one outputs a sentence. There's a modification to this called the Attention Model, that makes all this work much better. The a**ttention algorithm, the attention idea has been one of the most influential ideas in deep learning**.

Let's take a look at how that works. Get a very long French sentence like this. What we are asking this green encoder in your network to do is, to read in the whole sentence and then memorize the whole sentences and store it in the activations conveyed here. Then for the purple network, the decoder network till then generate the English translation. Jane went to Africa last September and enjoyed the culture and met many wonderful people; she came back raving about how wonderful her trip was, and is tempting me to go too. Now, the way a human translator would translate this sentence is not to first read the whole French sentence and then memorize the whole thing and then regurgitate an English sentence from scratch. Instead, what the human translator would do is read the first part of it, maybe generate part of the translation. Look at the second part, generate a few more words, look at a few more words, generate a few more words and so on.

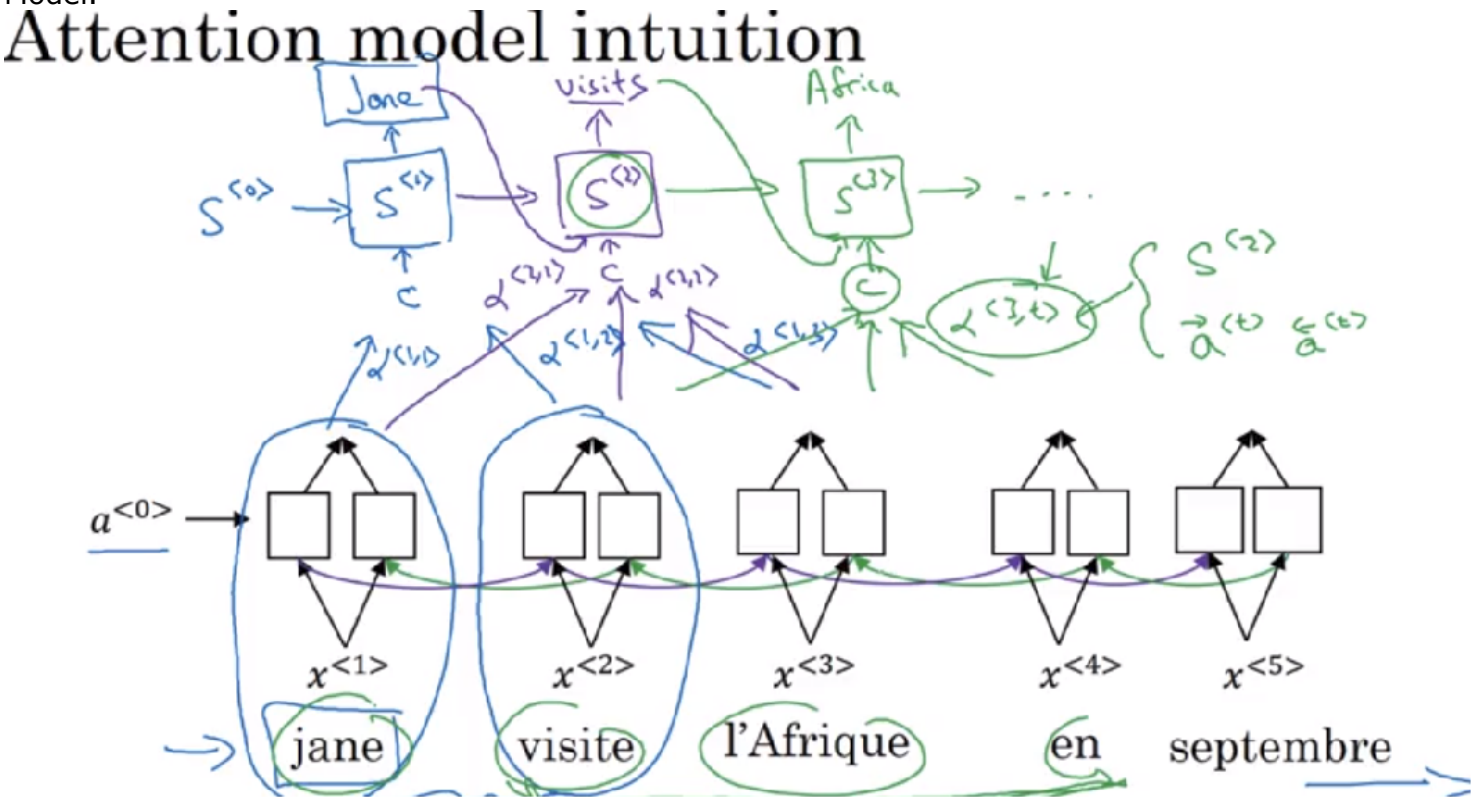


You kind of work part by part through the sentence, because it's just really difficult to memorize the whole long sentence like that. What you see for the Encoder-Decoder architecture above is that, it works quite well for short sentences, so we might achieve a relatively high Bleu score, but for very long sentences, maybe longer than 30 or 40 words, the performance comes down. The Bleu score might look like this as the sentence that varies and short sentences are just hard to translate, hard to get all the words, right? Long sentences, it doesn't do well on because it's just difficult to get in your

network to memorize a super long sentence. In this and the next section, you'll see the Attention Model which translates maybe a bit more like humans might, looking at part of the sentence at a time and with an Attention Model, machine translation systems performance can look like this, because by working one part of the sentence at a time, you don't see this huge dip which is really measuring the ability of a neural network to memorize a long sentence which maybe isn't what we most badly need a neural network to do. In this video, I want to just give you some intuition about how attention works and then we'll flesh out the details in the next video.

The Attention Model was due to Dimitri, Bahdanau, Camcrun Cho, Yoshe Bengio and even though it was obviously developed for machine translation, it spread to many other application areas as well. This is really a very influential, I think very seminal paper in the deep learning literature. Let's illustrate this with a short sentence, even though these ideas were maybe developed more for long sentences, but it'll be easier to illustrate these ideas with a simpler example. We have our usual sentence, Jane visite l'Afrique en Septembre. Let's say that we use a R and N, and in this case, I'm going to use a bidirectional R and N, in order to compute some set of features for each of the input words and you have to understand it, bidirectional R and N with outputs Y1 to Y3 and so on up to Y5 but we're not doing a word for word translation, let me get rid of the Y's on top. But using a bidirectional R and N, what we've done is for each other words, really for each of the five positions into sentence, you can compute a very rich set of features about the words in the sentence and maybe surrounding words in every position. Now, let's go ahead and generate the English translation. We're going to use another R and N to generate the English translations. Here's my R and N note as usual and instead of using A to denote the activation, in order to avoid confusion with the activations down here, I'm just going to use a different notation, I'm going to use S to denote the hidden state in this R and N up here, so instead of writing A1 I'm going to right S1 and so we hope in this model that the first word it generates will be Jane, to generate Jane visits Africa in September. Now, the question is, when you're trying to generate this first word, this output, what part of the input French sentence should you be looking at? Seems like you should be looking primarily at this first word, maybe a few other words close by, but you don't need to be looking way at the end of the sentence. What the Attention Model would be computing is a set of attention weights and we're going to use Alpha one, one to denote when you're generating the first words, how much should you be paying attention to this first piece of information here. And then we'll also come up with a second that's called Attention Weight, Alpha one, two which tells us what we're trying to compute the first work of Jane, how much attention we're paying to this second work from the inputs and so on and the Alpha one, three and so on, and together this will tell us what is exactly the context from denoter C that we should be paying attention to, and that is input to this R and N unit to then try to generate the first words. That's one step of the R and N, we will flesh out all these details in the next video. For the second step of this R and N, we're going to have a new hidden state S two and we're going to have a new set of the attention weights. We're going to have Alpha two, one to tell us when we generate in the second word. I guess this will be visits maybe that being the ground trip label. How much should we paying attention to the first word in the french input and also, Alpha two, two and so on. How much should we paying attention the word visite, how much should we pay attention to the free and so on. And of course, the first word we generate in Jane is also an input to this, and then we have some context that we're paying attention to and the second step, there's also an input and that together will generate the second word and that leads us to the third step, S three, where this is an input and we have some new context C that depends on the various Alpha three for the different time sets, that tells us how much should we be paying attention to the different words from the input French sentence and so on. So, some things I haven't specified yet, but that will go further into detail in the next video of this, how exactly this context defines and the goal of the context is for the third word is really should capture that maybe we should be looking around this part of the sentence. The formula you use to do that will defer to the next video as well as how do you compute these attention weights. And you see in the next video that Alpha three T, which is, when you're trying to generate the third word, I guess this would be the Africa, just getting the right output. The amounts that this R and N step should be paying attention to the French word that time T, that depends on the activations of the bidirectional R and N at time T, I guess it depends on the fourth activations and the, backward activations at time T and it will depend on the state from the previous steps, it will depend on S two, and these things together will influence, how much you pay attention to a specific word in the input French sentence. But we'll flesh out all these details in the next video. But the key intuition to take

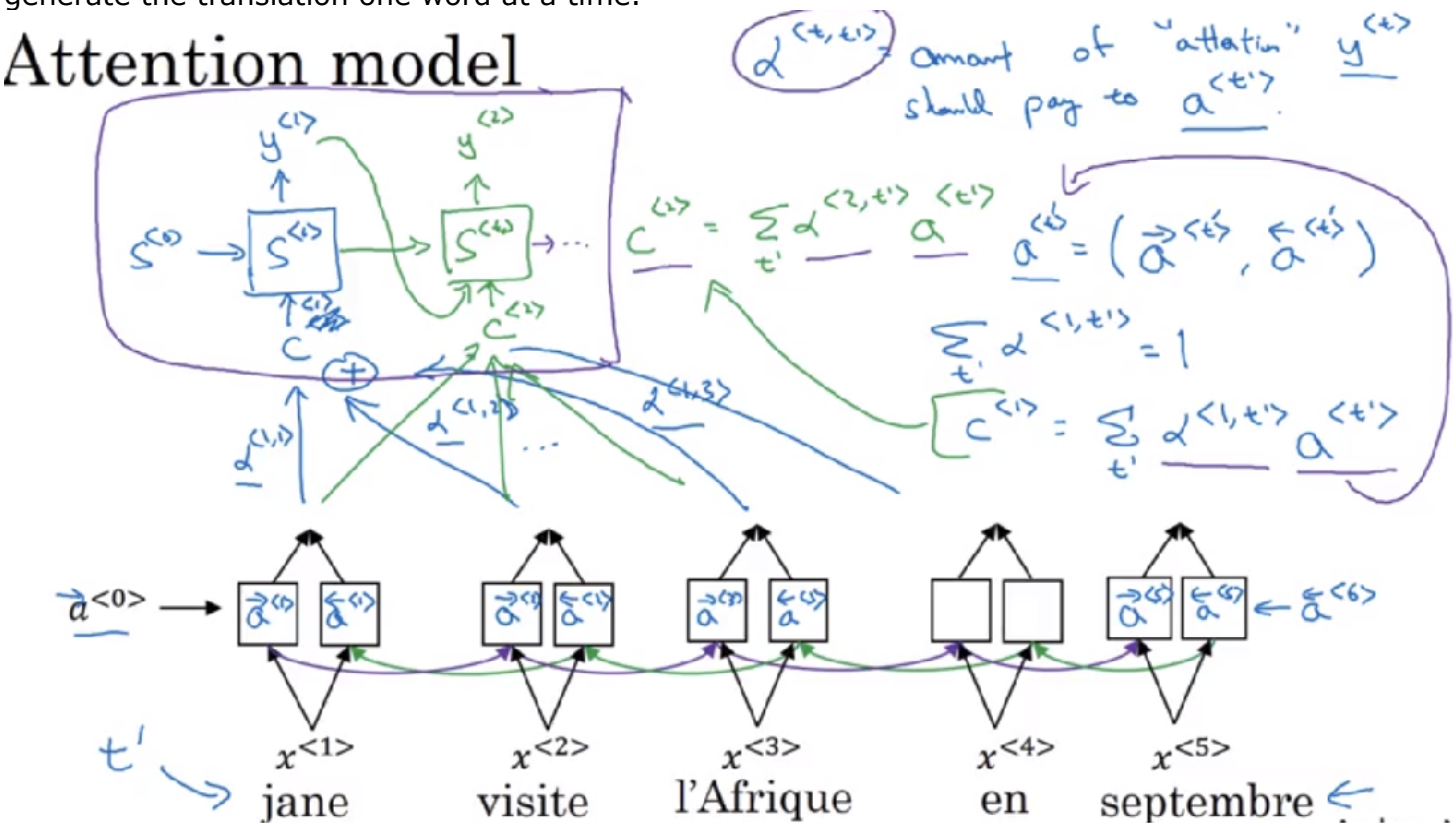
away is that this way the R and N marches forward generating one word at a time, until eventually it generates maybe the EOS and at every step, there are these attention weighs. Alpha T.T. Prime that tells it, when you're trying to generate the T, English word, how much should you be paying attention to the T prime French words. And this allows it on every time step to look only maybe within a local window of the French sentence to pay attention to, when generating a specific English word. I hope this video conveys some intuition about Attention Model and that we now have a rough sense of, maybe how the algorithm works. Let's go to the next video to flesh out the details of the Attention Model.



Attention Model

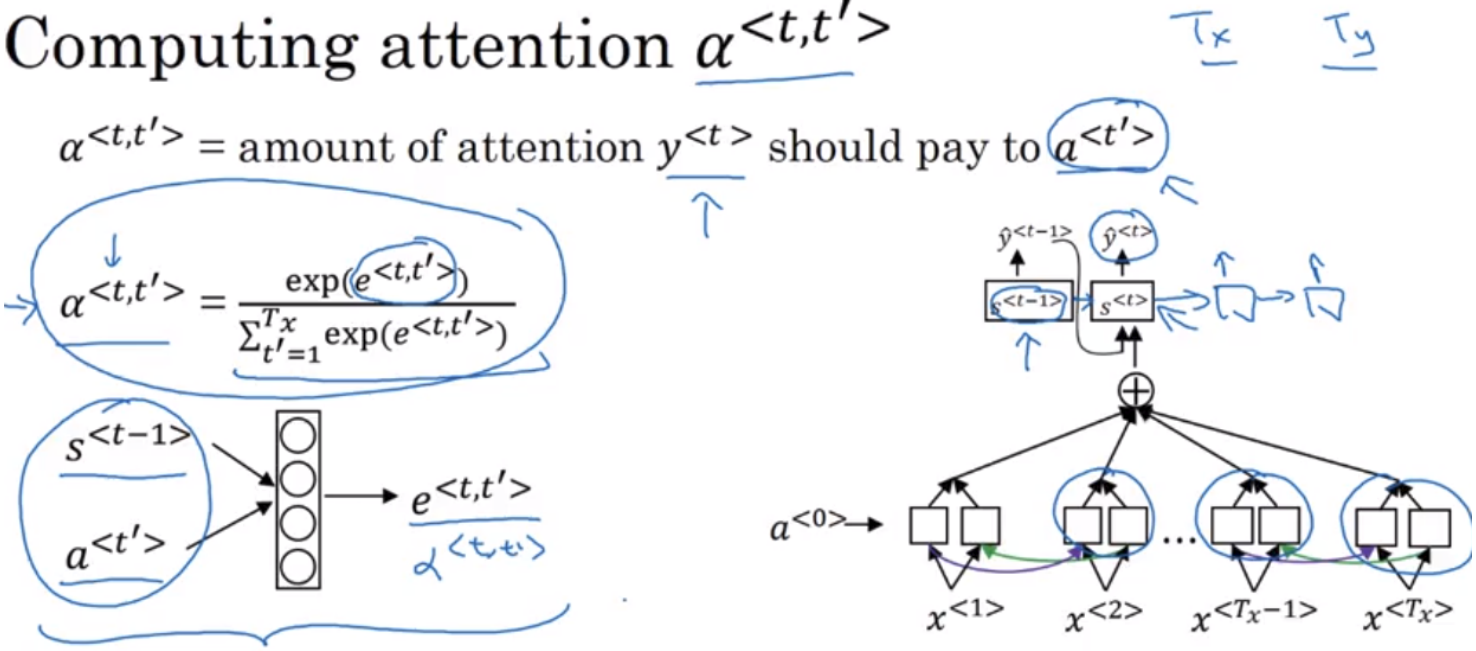
In the last video, you saw how the attention model allows a neural network to pay attention to only part of an input sentence while it's generating a translation, much like a human translator might. Let's now formalize that intuition into the exact details of how you would implement an attention model. So same as in the previous video, let's assume you have an input sentence and you use a bidirectional RNN, or bidirectional GRU, or bidirectional LSTM to compute features on every word. In practice, GRUs and LSTMs are often used for this, with maybe LSTMs be more common. And so for the forward occurrence, you have a forward occurrence first time step. Activation backward occurrence, first time step. Activation forward occurrence, second time step. Activation backward and so on. For all of them in just a forward fifth time step a backwards fifth time step. We had a zero here technically we can also have I guess a backwards sixth as a factor of all zero, actually that's a factor of all zeroes. And then to simplify the notation going forwards at every time step, even though you have the features computed from the forward occurrence and from the backward occurrence in the bidirectional RNN. I'm just going to use a of t to represent both of these concatenated together. So a of t is going to be a feature vector for time step t. Although to be consistent with notation, we're using second, I'm going to call this t_prime. Actually, I'm going to use t_prime to index into the words in the French sentence. Next, we have our forward only, so it's a single direction RNN with state s to generate the translation. And so the first time step, it should generate y1 and just will have as input some context C. And if you want to index it with time I guess you could write a C1 but sometimes I just right C without the superscript one. And this will depend on the attention parameters so alpha_11, alpha_12 and so on tells us how much attention. And so these alpha parameters tells us how much the context would depend on the features we're getting or the activations we're getting from the different time steps. And so the way we define the context is actually be a way to some of the features from the different time steps waited by these attention

waits. So more formally the attention waits will satisfy this that they are all be non-negative, so it will be a zero positive and they'll sum to one. We'll see later how to make sure this is true. And we will have the context or the context at time one often drop that superscript that's going to be sum over t_prime, all the values of t_prime of this waited sum of these activations. So this term here are the attention waits and this term here comes from here. So alpha(t_prime) is the amount of attention that's yt should pay to a of t_prime. So in other words, when you're generating the t of the output words, how much you should be paying attention to the t_primeth input to word. So that's one step of generating the output and then at the next time step, you generate the second output and is again done some of where now you have a new set of attention waits on they to find a new way to sum. That generates a new context. This is also input and that allows you to generate the second word. Only now just this way to sum becomes the context of the second time step is sum over t_prime alpha(2, t_prime). So using these context vectors. C1 right there back, C2, and so on. This network up here looks like a pretty standard RNN sequence with the context vectors as output and we can just generate the translation one word at a time.

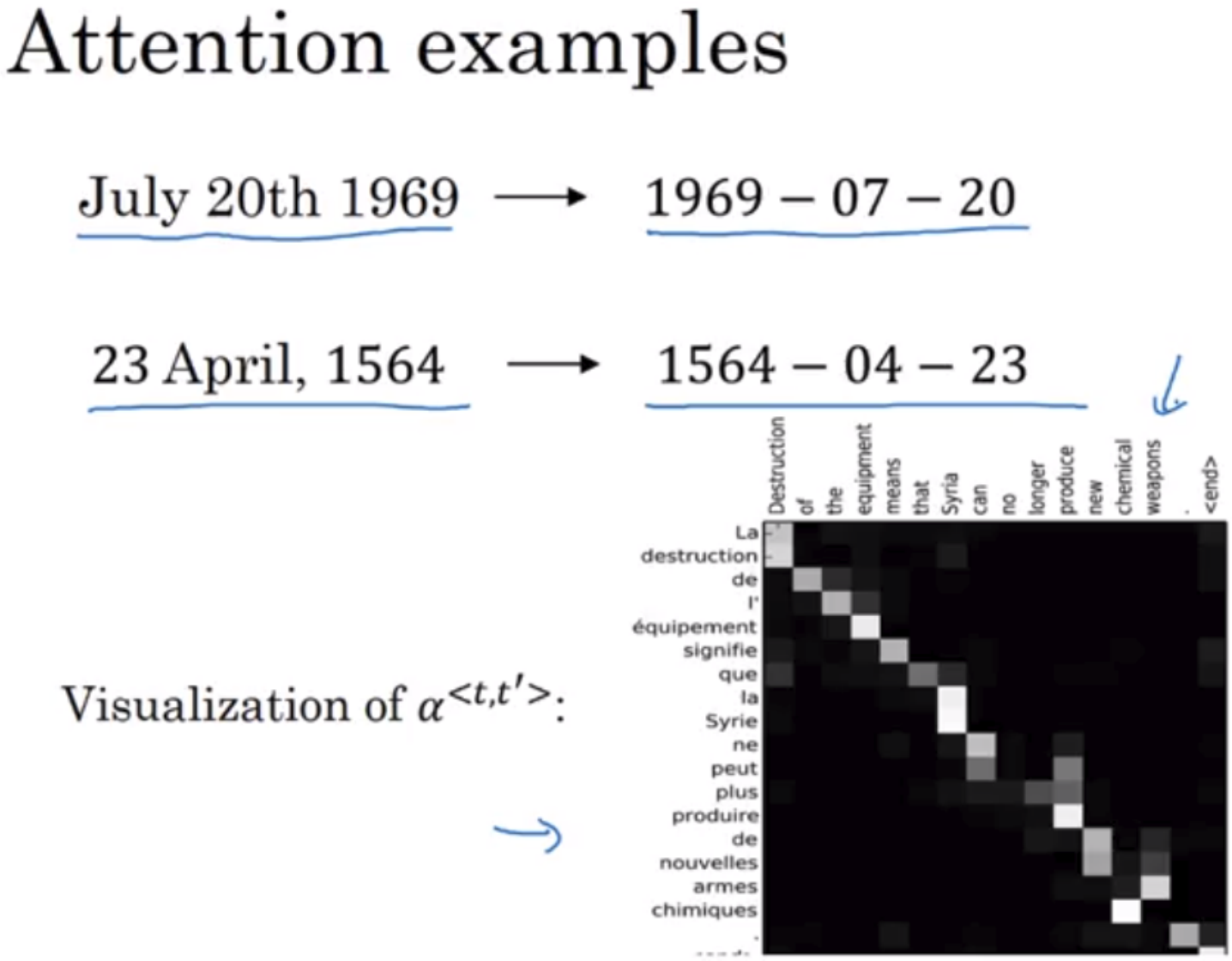


We have also define how to compute the context vectors in terms of these attention ways and those features of the input sentence. So the only remaining thing to do is to define how to actually compute these attention waits. Let's do that on the next slide. So just to recap, alpha(t, t_prime) is the amount of attention you should paid to a(t_prime) when you're trying to generate the t th words in the output translation. So let me just write down the formula and we talk of how this works. This is formula you could use the compute alpha(t, t_prime) which is going to compute these terms e(t, t_prime) and then use essentially a soft pass to make sure that these waits sum to one if you sum over t_prime. So for every fix value of t, these things sum to one if you're summing over t_prime. And using this soft max prioritization, just ensures this properly sums to one. Now how do we compute these factors e. Well, one way to do so is to use a small neural network as follows. So s t minus one was the neural network state from the previous time step. So here is the network we have. If you're trying to generate yt then st minus one was the hidden state from the previous step that just fell into st and that's one input to very small neural network. Usually, one hidden layer in neural network because you need to compute these a lot. And then a(t_prime) the features from time step t_prime is the other inputs. And the intuition is, if you want to decide how much attention to pay to the activation of t_prime. Well, the things that seems like it should depend the most on is what is your own hidden state activation from the previous time step. You don't have the current state

activation yet because of context feeds into this so you haven't computed that. But look at whatever you're hidden stages of this RNN generating the upper translation and then for each of the positions, each of the words look at their features. So it seems pretty natural that alpha(t, t_prime) and e(t, t_prime) should depend on these two quantities. But we don't know what the function is. So one thing you could do is just train a very small neural network to learn whatever this function should be. And trust that obligation trust wait and descent to learn the right function. And it turns out that if you implemented this whole model and train it with gradient descent, the whole thing actually works. This little neural network does a pretty decent job telling you how much attention yt should pay to a(t_prime) and this formula makes sure that the attention waits sum to one and then as you chug along generating one word at a time, this neural network actually pays attention to the right parts of the input sentence that learns all this automatically using gradient descent. Now, one downside to this algorithm is that it does take quadratic time or quadratic cost to run this algorithm. If you have tx words in the input and ty words in the output then the total number of these attention parameters are going to be tx times ty. And so this algorithm runs in quadratic cost.



Although in machine translation applications where neither input nor output sentences is usually that long maybe quadratic cost is actually acceptable. Although, there is some research work on trying to reduce costs as well. Now, so far up in describing the attention idea in the context of machine translation. Without going too much into detail this idea has been applied to other problems as well. So just image captioning. So in the image capturing problem the task is to look at the picture and write a caption for that picture. So in this paper set to the bottom by Kevin Chu, Jimmy Barr, Ryan Kiros, Kelvin Shaw, Aaron Korver, Russell Zarkutnov, Virta Zemo, and Andrew Benjo they also showed that you could have a very similar architecture. Look at the picture and pay attention only to parts of the picture at a time while you're writing a caption for a picture. So if you're interested, then I encourage you to take a look at that paper as well. And you get to play with all this and more in the programming exercise. Whereas machine translation is a very complicated problem in the prior exercise you get to implement and play of the attention while you yourself for the date normalization problem. So the problem inputting a date like this. This actually has a date of the Apollo Moon landing and normalizing it into standard formats or a date like this and having a neural network a sequence, sequence model normalize it to this format. This by the way is the birthday of William Shakespeare.



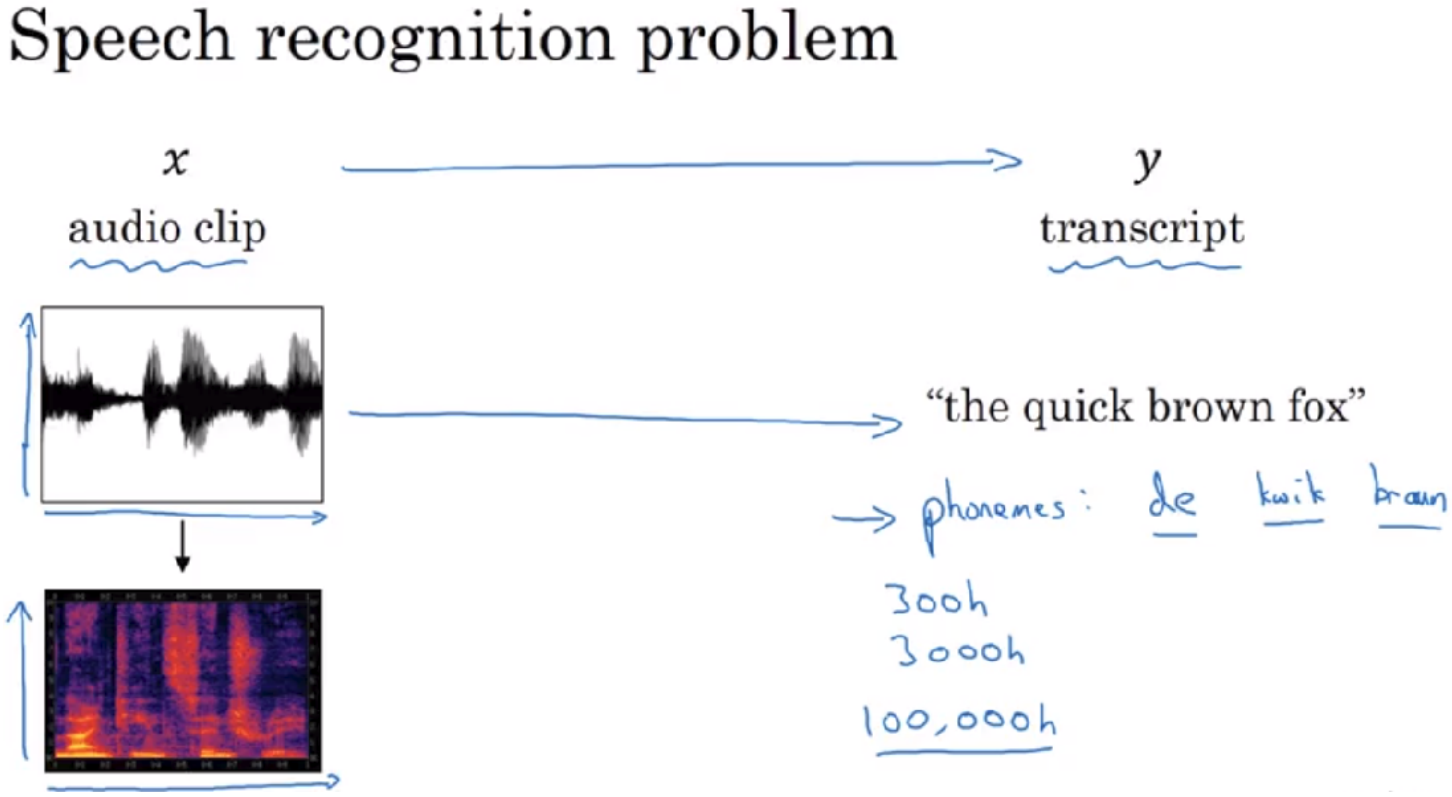
Also it's believed to be. And what you see in prior exercises as you can train a neural network to input dates in any of these formats and have it use an attention model to generate a normalized format for these dates. One other thing that sometimes fun to do is to look at the visualizations of the attention waits. So here's a machine translation example and here were plotted in different colors. the magnitude of the different attention waits. I don't want to spend too much time on this but you find that the corresponding input and output words you find that the attention waits will tend to be high. Thus, suggesting that when it's generating a specific word in output is, usually paying attention to the correct words in the input and all this including learning where to pay attention when was all learned using propagation with an attention model.

Speech Recognition - Audio data

Speech recognition

One of the most exciting developments were sequence-to-sequence models has been the rise of very accurate speech recognition. We're nearing the end of the course, we want to take just a couple of videos to give you a sense of how these sequence-to-sequence models are applied to audio data, such as the speech. So, what is the speech recognition problem? You're given an audio clip, x, and your job is to automatically find a text transcript, y. So, an audio clip, if you plot it looks like this, the horizontal axis here is time, and what a microphone does is it really measures minuscule changes in air pressure, and the way you're hearing my voice right now is that your ear is detecting little changes in air pressure, probably generated either by your speakers or by a headset. And some audio clips like this plots with the air pressure against time. And, if this audio clip is of me saying, "the quick brown fox", then hopefully, a speech recognition algorithm can input that audio clip and output that transcript. And because even the human ear doesn't process raw wave forms, but the human ear

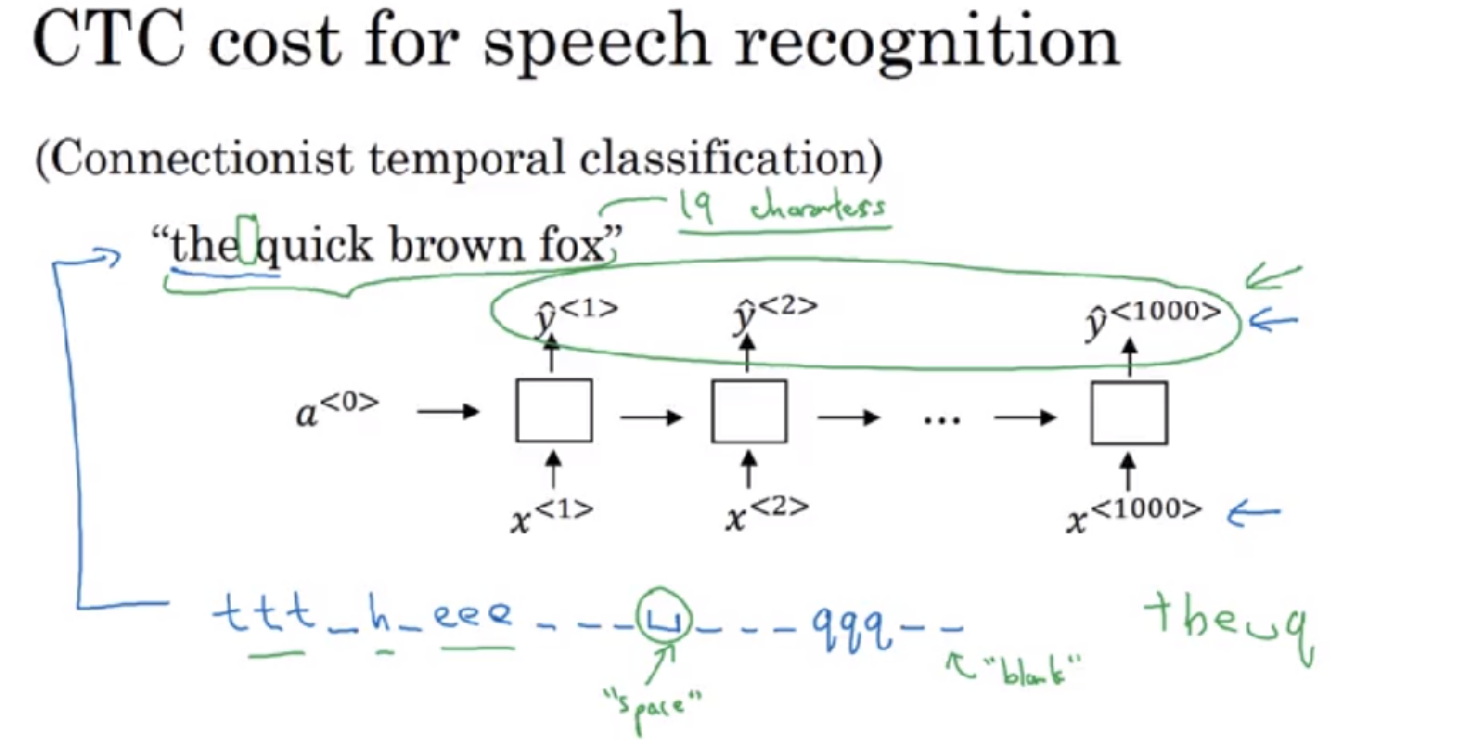
has physical structures that measures the amounts of intensity of different frequencies, there is, a common pre-processing step for audio data is to run your raw audio clip and generate a spectrogram. So, this is the plots where the horizontal axis is time, and the vertical axis is frequencies, and intensity of different colors shows the amount of energy. So, how loud is the sound at different frequencies? At different times? And so, these types of spectrograms, or you might also hear people talk about false back outputs, is often commonly applied pre-processing step before audio is pass into in the running algorithm. And the human ear does a computation pretty similar to this pre-processing step. So, one of the most exciting trends in speech recognition is that, once upon a time, speech recognition systems used to be built using phonemes and this where, I want to say hand-engineered basic units of cells.



So, the quick brown fox represented as phonemes. I'm going to simplify a bit, let say, "The" has a "de" and "e" sound and Quick, has a "ku" and "wu", "ik", "k" sound, and linguist used to write off these basic units of sound, and try the Greek language down to these basic units of sound. So, brown, this aren't the official phonemes which are written with more complicated notation, but linguists use to hypothesize that writing down audio in terms of these basic units of sound called phonemes would be the best way to do speech recognition. But with end-to-end deep learning, we're finding that phonemes representations are no longer necessary. But instead, you can built systems that input an audio clip and directly output a transcript without needing to use hand-engineered representations like these. One of the things that made this possible was going to much larger data sets. So, academic data sets on speech recognition might be as a 300 hours, and in academia, 3000 hour data sets of transcribed audio would be considered reasonable size, so lot of research has been done, a lot of research papers that are written on data sets there are several thousand voice. But, the best commercial systems are now trains on over 10,000 hours and sometimes over a 100,000 hours of audio. And, it's really moving to a much larger audio data sets, transcribe audio data sets were both x and y, together with deep learning algorithm, that has driven a lot of progress is speech recognition.

So, how do you build a speech recognition system? In the last section, we're talking about the attention model. So, one thing you could do is actually do that, where on the horizontal axis, you take in different time frames of the audio input, and then you have an attention model try to output the transcript like, "the quick brown fox", or what it was said. One other method that seems to work well is to use the CTC cost for speech recognition. CTC stands for **Connectionist Temporal Classification** and is due to Alex Graves, Santiago Fernandes, Faustino Gomez, and Jürgen Schmidhuber. So, here's the idea. Let's say the audio clip was someone saying, "the quick brown

fox". We're going to use a neural network structured as shown in diagram below with an equal number of input x's and output y's, and we have drawn a simple of what uni-directional for the RNN for this, but in practice, this will usually be a bidirectional LSTM and bidirectional GRU and usually, a deeper model. But notice that the number of time steps here is very large and in speech recognition, usually the number of input time steps is much bigger than the number of output time steps. So, for example, if you have 10 seconds of audio and your features come at a 100 hertz so 100 samples per second, then a 10 second audio clip would end up with a thousand inputs. Right, so it's 100 hertz times 10 seconds, and so with a thousand inputs. But your output might not have a thousand alphabets, might not have a thousand characters. So, what do you do? The CTC cost function allows the RNN to generate an output like this ttt, there's a special character called the blank character, which we're going to write as an underscore here, h_eee___, and then maybe a space, we're going to write like this, so that a space and then ___ qqq__. And, this is considered a correct output for the first parts of the space, quick with the Q, and the basic rule for the CTC cost function is to collapse repeated characters not separated by "blank". So, to be clear, I'm using this underscore to denote a special blank character and that's different than the space character. So, there is a space here between the and quick, so I should output a space. But, by collapsing repeated characters, not separated by blank, it actually collapse the sequence into t, h, e, and then space, and q, and this allows your network to have a thousand outputs by repeating characters allow the times. So, inserting a bunch of blank characters and still ends up with a much shorter output text transcript. So, this phrase here **"the quick brown fox"** including spaces actually has 19 characters, and if somehow, the newer network is forced upwards of a thousand characters by allowing the network to insert blanks and repeated characters and can still represent this 19 character upwards with this 1000 outputs of values of Y.

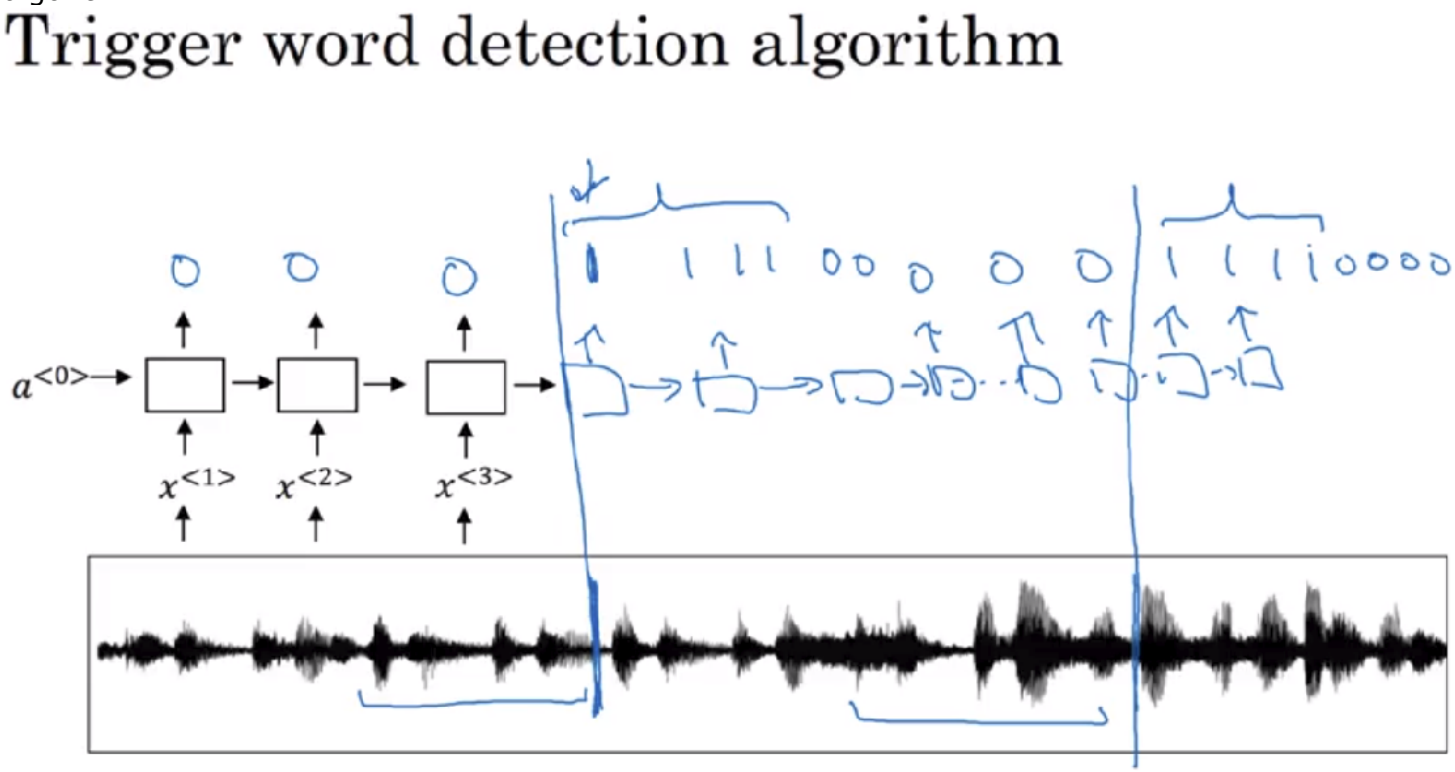


So, this paper by Alex Grace, as well as by those deep speech recognition system, which I was involved in, used this idea to build effective Speech recognition systems. So, I hope that gives you a rough sense of how speech recognition models work. Attention like models work and CTC models work and present two different options of how to go about building these systems. Now, today, building effective where production skills speech recognition system is a pretty significant effort and requires a very large data set. But, what I like to do in the next video is share you, how you can build a trigger word detection system, where keyword detection system which is actually much easier and

can be done with even a smaller or more reasonable amount of data. So, let's talk about that in the next video.

Trigger Word Detection

We've now learned so much about deep learning and sequence models that we can actually describe a trigger word system quite simply but when the rise of speech recognition have been more and more devices you can wake up with your voice and those are sometimes called **trigger word detection** systems so let's see how you can build a trigger word system examples of triggering systems include Amazon echo which is broken out what that word alexa or how Apple Siri working out with hey Siri and Google home woken up with ok Google so stands the trigger word detection that if you have say an Amazon echo in your living room you can walk the living room and just say Alexa what time is it and have it wake up I'll be triggered by the words of alexa and answer your voice query so if you can build a trigger word detection system maybe you can make your computer do something by telling it computer activate one of my friends also works on turning on an offer particular lamp using a trigger word kind of as a fun project but what I want to show you is how you can build a trigger word detection system now the trigger word detection literature is still evolving so there actually isn't a single universally agreed on algorithm for trigger word detection yet the literature on trigger word detection algorithm is still evolving so there isn't wide consensus yet on what's the best algorithm for trigger word detection so I'm just going to show you one example of an algorithm.



You can use now you've seen our ends like this and what we really do is take an audio clip maybe compute spectrogram features and that generates features x1 x2 x3 audio features x1 x2 x3 that you pass through an RNN and so all that remains to be done is to define the target labels Y so if this point in the audio clip is when someone just finished saying the trigger word such as hey Siri or okay Google then in the training sets you can set the target labels to be zero for everything before that point and right after that to set the target label of one and then if a little bit later on you know the trigger word was set again and the trigger was said at this point then you can again set the target label to be one right after that now this type of labeling scheme for an RNN you know could work actually this won't actually work reasonably well one slight disadvantage of this is it creates a very imbalanced training set so if a lot more zeros than ones so one other thing you could do that it's getting a little bit of a hack but could make them all the little bit easy to train is instead of setting only a single time step to output one you can actually make an output a few ones for several times or for a fixed period of time before reverting back to zero so and that thumb slightly evens out the ratio of ones to zeros but this is a little bit of a hack but if this is when in the audio clipper trigger where

the set then right after that you can set the toggle able to one and if this is the trigger words say the game then right after that just when you want the RNN to output one but so I think you should feel quite proud of yourself we've learned enough about the learning that it just takes one picture at one slide to this to describe something as complicated as trigger word detection and based on this I hope you'd be able to implement something that works and allows you to detect trigger words and get it to work maybe even make it do something fun in your house that I'm like turn on or turn off you could do something like a computer when you're when someone else says they trigger words on this is the last technical section of this course and to wrap up in this course on sequence models you learned about RNN's including both GRU and LSTM and then in the second week you learned a lot about word embeddings and how they learn representations of words and then in this week you learned about the attention model as well as how to use it to process audio data.

**********END OF COURSE **********