**Course 5: Sequence Models**

Study Notes by Jane Huang 08/06/2019

Credit to Andrew NG

Table of Contents

[Week 1 1](#_Toc16019908)

[1.1 Why sequence models 1](#_Toc16019909)

[1.2 Notation 1](#_Toc16019910)

[1.3 RNN model 2](#_Toc16019911)

[1.4 Back Prop through Time 3](#_Toc16019912)

[1.5 Different Types of RNNs 4](#_Toc16019913)

[1.6 Language Model and sequence generation 5](#_Toc16019914)

[1.7 Sampling Novel Sequences 6](#_Toc16019915)

[1.8 Vanishing Gradient with RNNs 7](#_Toc16019916)

[1.9 GRU (Gated Recurrent Unit) 8](#_Toc16019917)

[1.10 LSTM (long short term memory) unit 9](#_Toc16019918)

[1.11 Bidirectional RNN 10](#_Toc16019919)

[1.12 Deep RNNs 10](#_Toc16019920)

[Week 2 11](#_Toc16019921)

[2.1 Word representation 11](#_Toc16019922)

[2.2 Using Word Embeddings 12](#_Toc16019923)

[2.3 Properties of Word Embeddings 13](#_Toc16019924)

[2.4 Embedding Matrix 14](#_Toc16019925)

[2.5 Negative Sampling 14](#_Toc16019926)

[2.6 GloVe word vectors 15](#_Toc16019927)

[2.7 Sentiment Classification 16](#_Toc16019928)

[2.8 Debiasing Word Embeddings 17](#_Toc16019929)

[Week 3 18](#_Toc16019930)

[3.1 Basic Models 18](#_Toc16019931)

[3.2 Picking the most likely sentences 19](#_Toc16019932)

[3.3 Beam Search 20](#_Toc16019933)

[3.4 Refinement to Beam search 21](#_Toc16019934)

[3.5 Error analysis in beam search 21](#_Toc16019935)

[3.6 Bleu Score 22](#_Toc16019936)

[3.7 Attention Model Intuition 24](#_Toc16019937)

[3.8 Attention Model 25](#_Toc16019938)

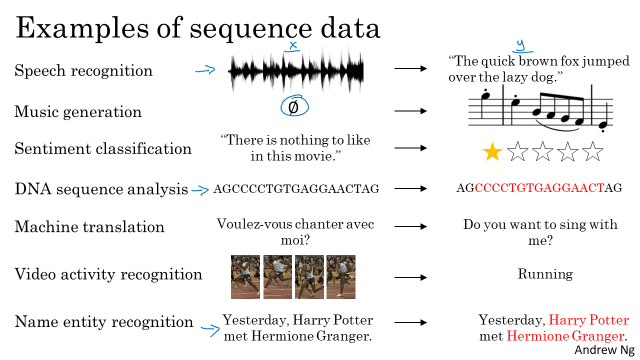
[3.9 Speech recognition 26](#_Toc16019939)

[3.10 Trigger Word Detection 27](#_Toc16019940)

[3.11 Summary and Thank You 27](#_Toc16019941)

**Week 1**

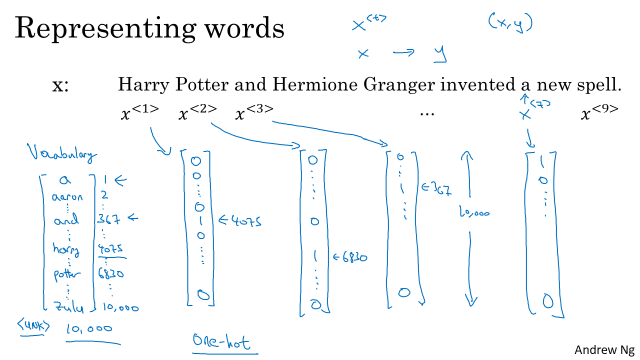
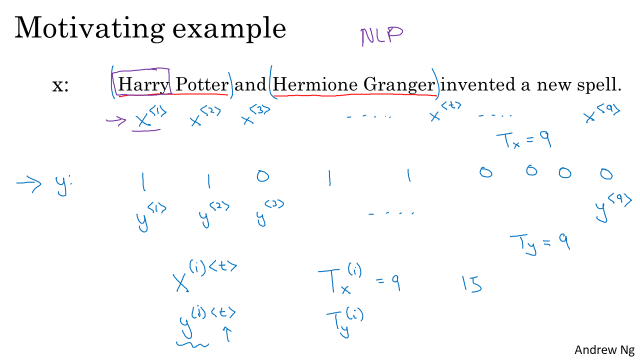
* 1. **Why sequence models**

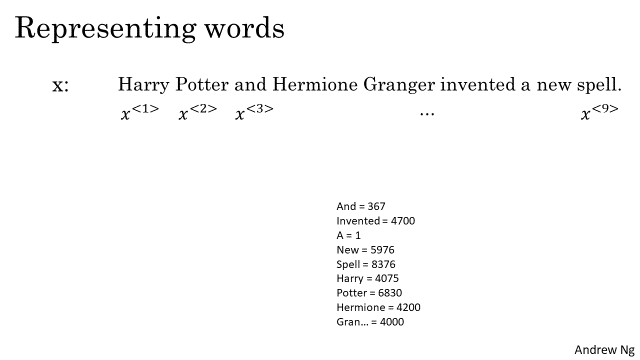


Music generation: only the output Y is a sequence, the input can be the empty set, or it can be a single integer, maybe referring to the genre of music you want to generate or maybe the first few notes of the piece of music you want.

* 1. **Notation**

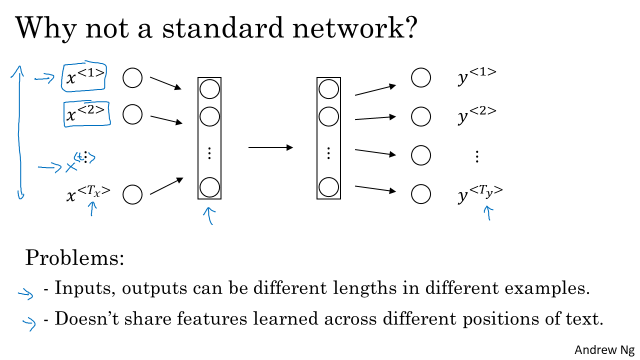
Let’s say you want a sequence model to automatically tell you where are the people’s names in this sentence. I'm going to use X\_t with the index t to index into positions, in the middle of the sequence. And t implies that these are temporal sequences although whether the sequences are temporal one or not, I'm going to use the index t to index into the positions in the sequence. And  Tx\_i would be the input sequence length for training example i.



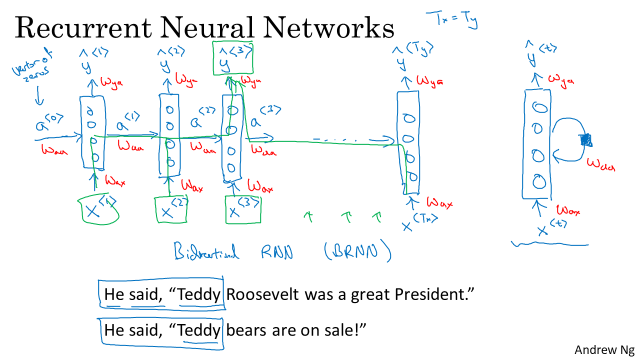


What if you encounter a word that is not in your vocabulary? Well the answer is, you create a new token or a new fake word called Unknown Word which under note as follows and go back as UNK to represent words not in your vocabulary.

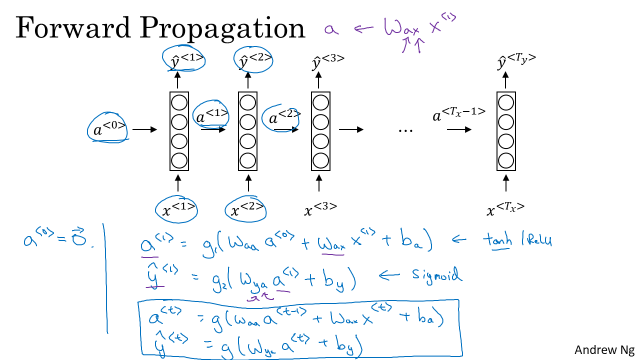
* 1. **RNN model**

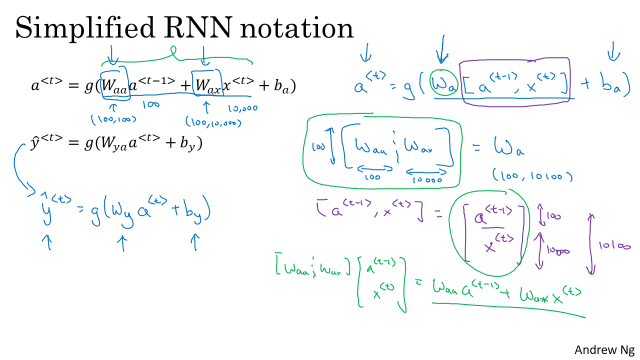


What is a recurrent neural network? Let's build one up. So if you are reading the sentence from left to right, the first word you will read is the some first words say X1, and what we're going to do is take the first word and feed it into a neural network layer. I'm going to draw it like this. So there's a hidden layer of the first neural network and we can have the neural network maybe try to predict the output. So is this part of the person's name or not. And what a recurrent neural network does is, when it then goes on to read the second word in the sentence, say x2, instead of just predicting y2 using only X2, it also gets to input some information from whether the computer that time step one. So in particular, deactivation value from time step one is passed on to time step two. Then at the next time step, recurrent neural network inputs the third word X3 and it tries to output some prediction, Y hat three and so on up until the last time step where it inputs x\_TX and then it outputs y\_hat\_ty.

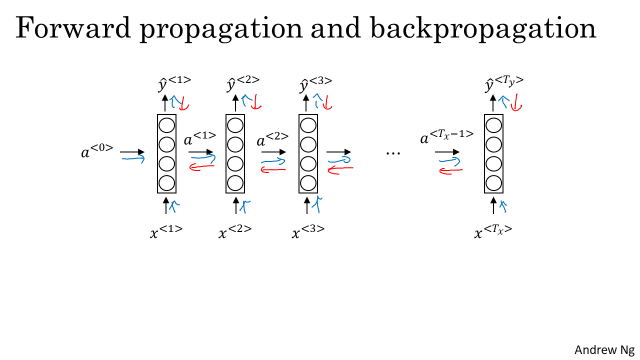


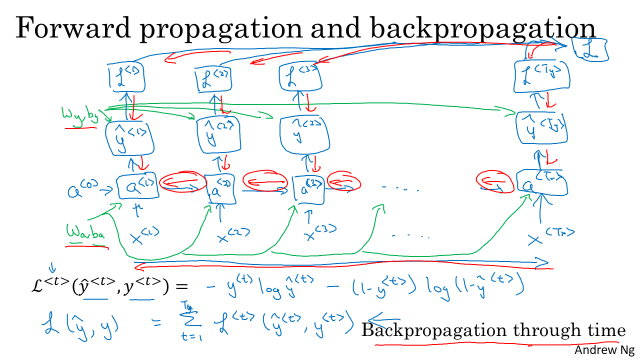
Now, one weakness of this RNN is that it only uses the information that is earlier in the sequence to make a prediction. In particular, when predicting y3, it doesn't use information about the worst X4, X5, X6 and so on. So this is a problem because if you are given a sentence, "He said Teddy Roosevelt was a great president." In order to decide whether or not the word Teddy is part of a person's name, it would 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 have been, "He said teddy bears they're on sale." So given just the first three words is 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, it 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 bi-directional recurrent neural networks or BRNNs.



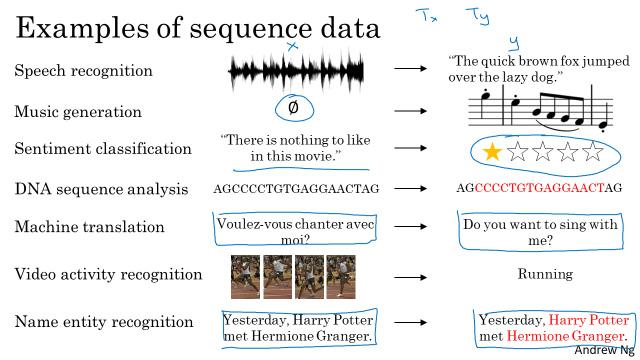


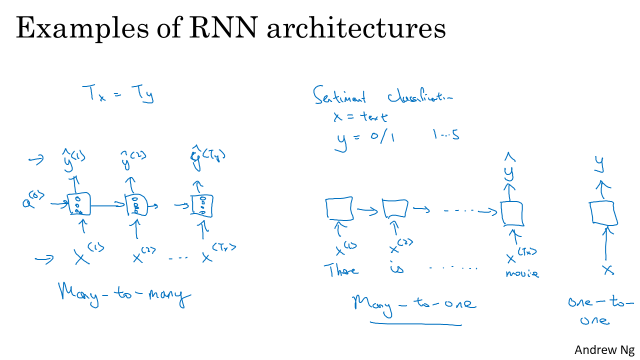
* 1. **Back Prop through Time**



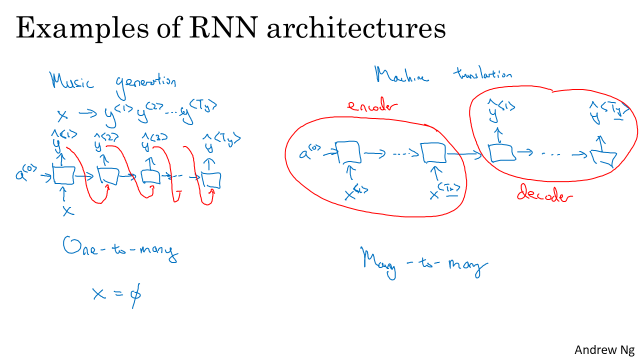


* 1. **Different Types of RNNs**





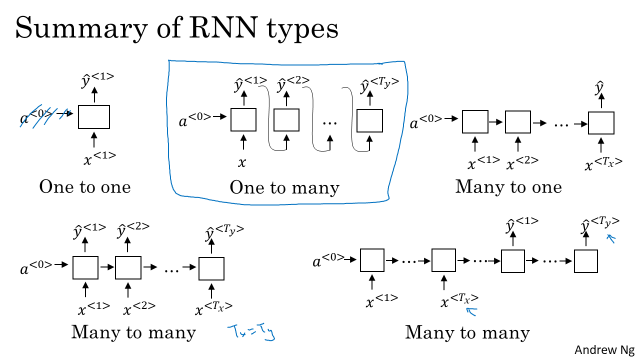
Sentiment classification: Many to one



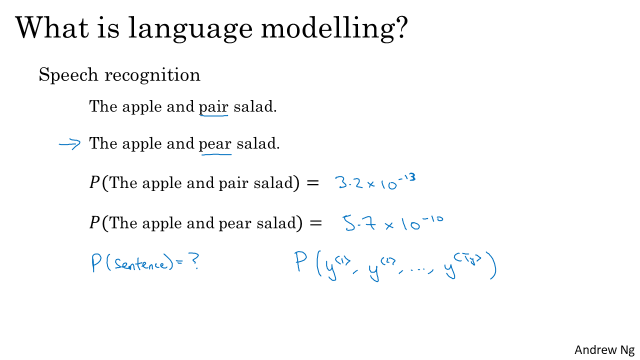
Music generation: one to many

Many to many (input length != output legth): machine translation, ex. French to English

One to one: we don’t have to use RNN~~



* 1. **Language Model and sequence generation**



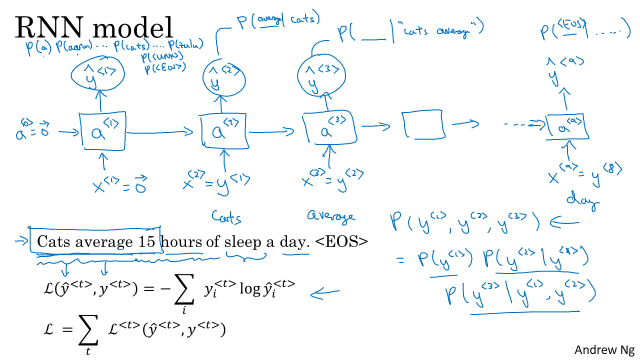
A statistical language model is a [probability distribution](https://en.wikipedia.org/wiki/Probability_distribution) over sequences of words. Given such a sequence, say of length *m*, it assigns a probability P(w1,w2,…,wm) to the whole sequence.

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?

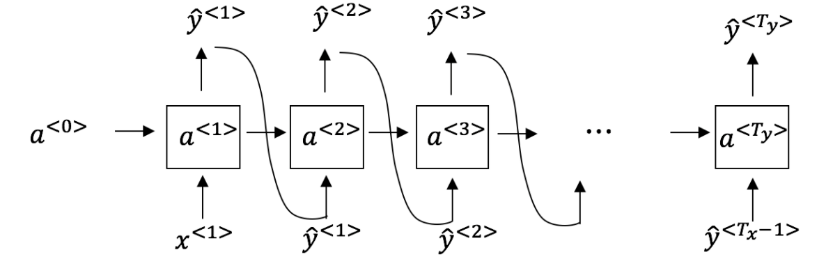


EOS: end of sentence

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.

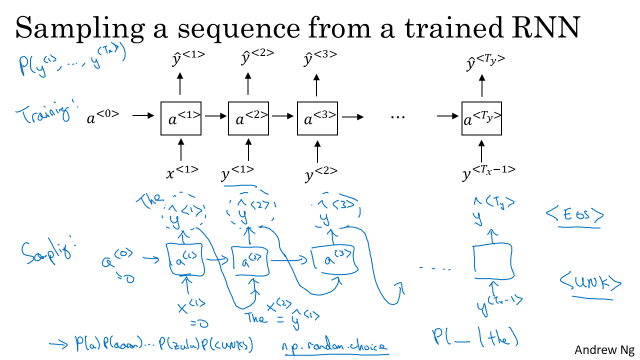


You have finished training a language model RNN and are using it to sample random sentences, as follows:

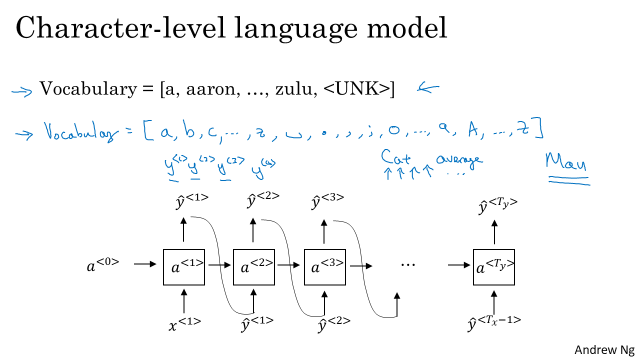


What are you doing at each time step *t*? (i) Use the probabilities output by the RNN to randomly sample a chosen word for that time-step as *y*^<*t*>. (ii) Then pass this selected word to the next time-step.

* 1. **Sampling Novel Sequences**

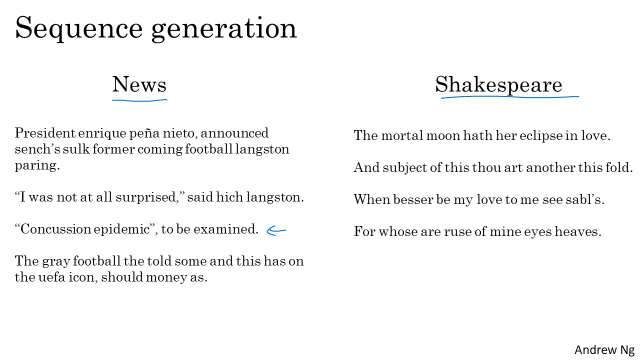


This particular procedure will sometimes generate an unknown word token. If you want to make sure that your algorithm never generates this token, one thing you could do is just reject any sample that came out as unknown word token and just keep resampling from the rest of the vocabulary until you get 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.



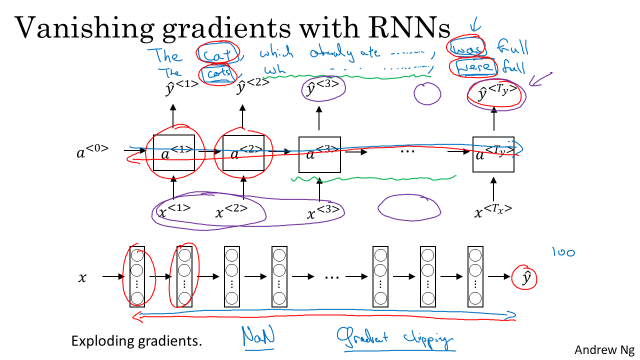
**Character level vs Word level**

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 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.



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.

* 1. **Vanishing Gradient with RNNs**



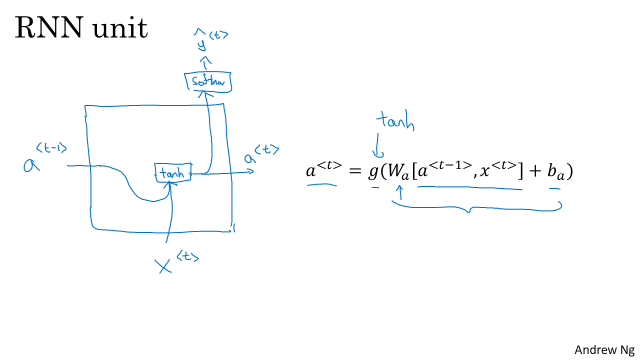
**Vanishing Gradients**

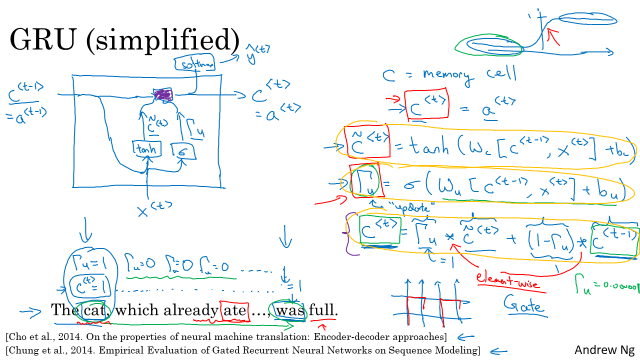
The **basics RNN we've seen so far it's not very good at capturing very long-term dependencies**. To explain why, you might remember from our early discussions of training very deep neural networks, that we talked about the vanishing gradients problem. So this is a very, very deep neural network say, 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.

**Exploding gradients 🡪 Gradient Clipping**

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. Varnishing gradient is much harder to solve than exploding gradient.

* 1. **GRU (Gated Recurrent Unit)**



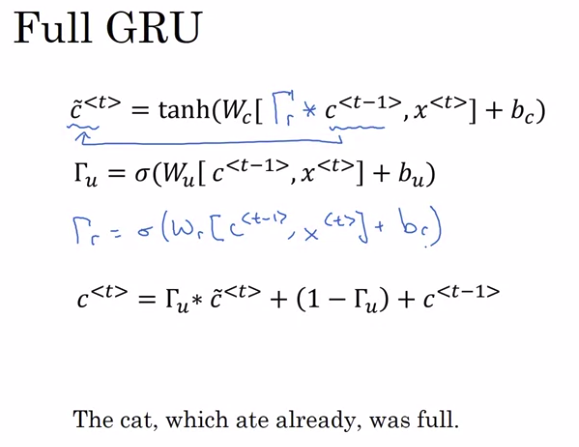


**C= memory cell**

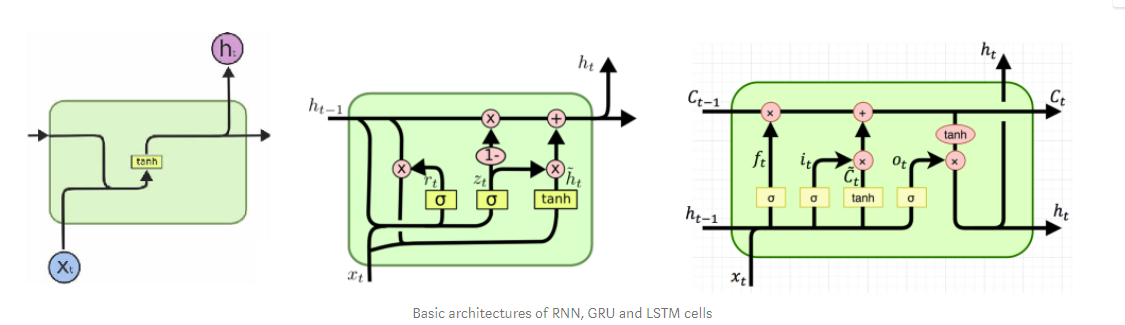
a: output activation value

update gates: gamma. Sigmoid function is used. But in most cases, it is either very big or very small, so it is more likely to be close to 1 or 0

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 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.

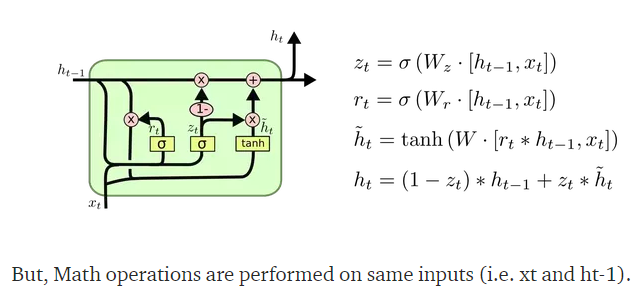


R: relevant (This gate gamma r tells you how relevant is computing the next candidate for c\_t)



**Simple RNN** :- Here there is simple multiplication of Input (xt) and Previous Output (ht-1). Passed through Tanh activation function. No Gates present.

**Gated Recurrent Unit (GRU) :-**Here a **Update gate**is introduced, to decide whether to pass Previous O/P (ht-1)to next Cell (as ht) or not. Forget gate is nothing but additional Mathematical Operations with a new set of Weights (Wt).



**Question: Why do many recurrent NNs use tanh and sigmoid?**

* The gate recurrent connections use sigmoid, and the cell recurrent connections use tanh. The cell recurrent connection need a function whose second derivative sustain for a long span to address the vanishing gradient problem. The gate recurrent connections could also use a such a function, but since they control the error flow, in both positive and negative way, they use sigmoid as non-linearity.
* LSTMs manage an internal state vector whose values should be able to increase or decrease when we add the output of some function. Sigmoid output is always non-negative; values in the state would only increase. The output from tanh can be positive or negative, allowing for increases and decreases in the state. That's why tanh is used to determine candidate values to get added to the internal state. The GRU cousin of the LSTM doesn't have a second tanh, so in a sense the second one is not necessary.
  1. **LSTM (long short term memory) unit**

**Core Concept:** The core concept of LSTM’s are the cell state, and it’s various gates. The cell state acts as a transport highway that transfers relative information all the way down the sequence chain. You can think of it as the “memory” of the network. The cell state, in theory, can carry relevant information throughout the processing of the sequence. So even information from the earlier time steps can make it’s way to later time steps, reducing the effects of short-term memory. As the cell state goes on its journey, information get’s added or removed to the cell state via gates. The gates are different neural networks that decide which information is allowed on the cell state. The gates can learn what information is relevant to keep or forget during training.



* **LSTM: update gate, forget gate, output gate**
* **Forget gate**

First, we have the forget gate. This gate decides what information should be thrown away or kept. Information from the previous hidden state and information from the current input is passed through the sigmoid function. Values come out between 0 and 1. The closer to 0 means to forget, and the closer to 1 means to keep

* **Input Gate**

To update the cell state, we have the input gate. First, we pass the previous hidden state and current input into a sigmoid function. That decides which values will be updated by transforming the values to be between 0 and 1. 0 means not important, and 1 means important. You also pass the hidden state and current input into the tanh function to squish values between -1 and 1 to help regulate the network. Then you multiply the tanh output with the sigmoid output. The sigmoid output will decide which information is important to keep from the tanh output.

* **Output Gate**

Last we have the output gate. The output gate decides what the next hidden state should be. Remember that the hidden state contains information on previous inputs. The hidden state is also used for predictions. First, we pass the previous hidden state and the current input into a sigmoid function. Then we pass the newly modified cell state to the tanh function. We multiply the tanh output with the sigmoid output to decide what information the hidden state should carry. The output is the hidden state. The new cell state and the new hidden is then carried over to the next time step.

* **Cell State**

Now we should have enough information to calculate the cell state. First, the cell state gets pointwise multiplied by the forget vector. This has a possibility of dropping values in the cell state if it gets multiplied by values near 0. Then we take the output from the input gate and do a pointwise addition which updates the cell state to new values that the neural network finds relevant. That gives us our new cell state.

* **GRU two dates: update gate, reset gate**
* Update Gate

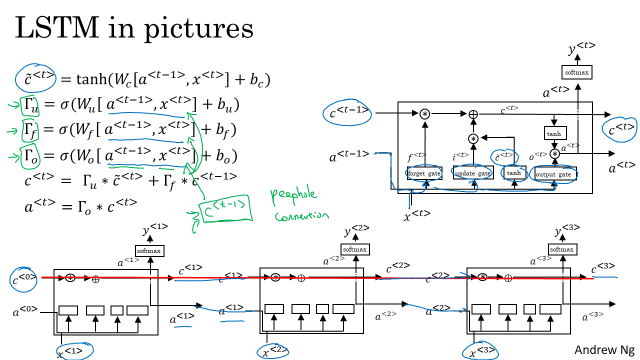
The update gate acts similar to the forget and input gate of an LSTM. It decides what information to throw away and what new information to add.

* Reset Gate

The reset gate is another gate is used to decide how much past information to forget.

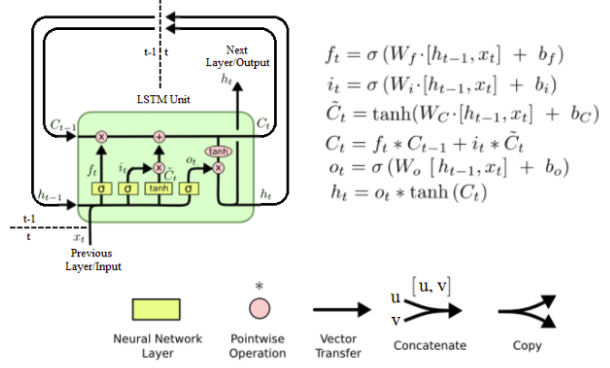
 GRU’s has fewer tensor operations; therefore, they are a little speedier to train then LSTM’s. There isn’t a clear winner which one is better. Researchers and engineers usually try both to determine which one works better for their use case.

To sum this up, RNN’s are good for processing sequence data for predictions but suffers from short-term memory. LSTM’s and GRU’s were created as a method to mitigate short-term memory using mechanisms called gates. Gates are just neural networks that regulate the flow of information flowing through the sequence chain. LSTM’s and GRU’s are used in state of the art deep learning applications like speech recognition, speech synthesis, natural language understanding, etc.



Peehole Connection: What that means is that the gate values may depend not just on 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.

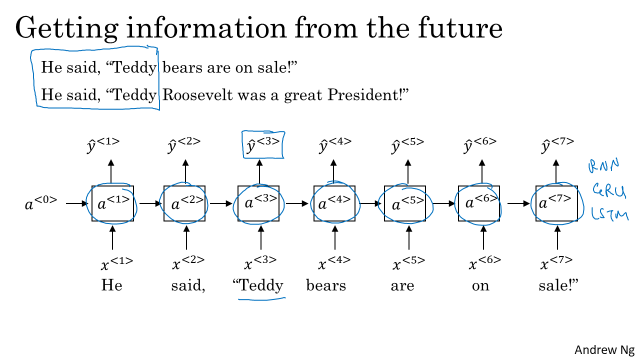
**Long Short Term Memory Unit (LSTM) :-**Here 2 more Gates are introduced (Forget and Output) in addition to Update gate of GRU. And again as above, these are additional Mathematical Operations on same inputs (xt and ht-1). So overall, LSTM has introduced 2 Math operations having 2 new sets of Weights.

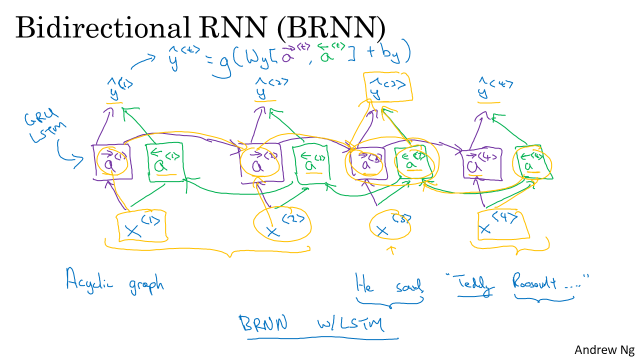


**GRU vs LSTM**

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.

* 1. **Bidirectional RNN**

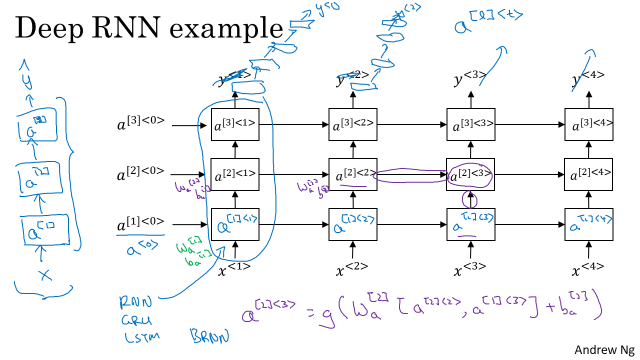




Acyclic Graph

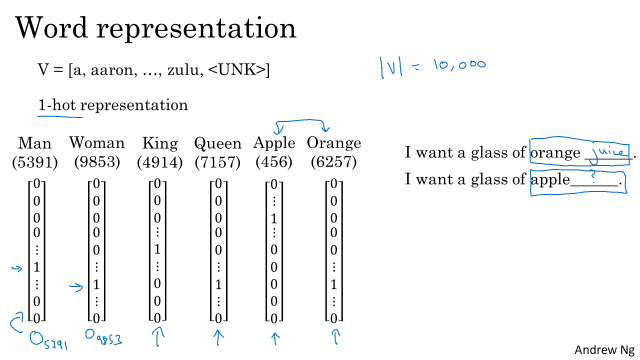
* 1. **Deep RNNs**

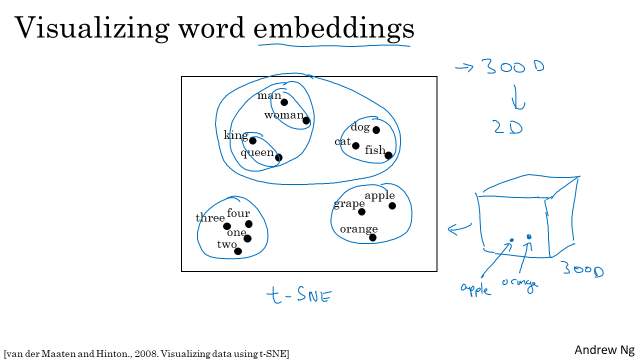
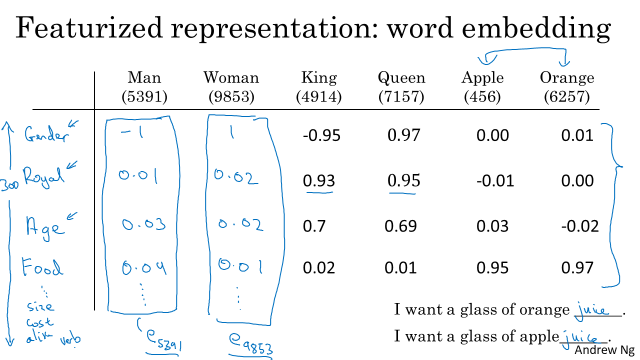
We're going to use is to denote that it's an activation associated with layer l and then <t> to denote that that's associated over time t.



**Week 2**

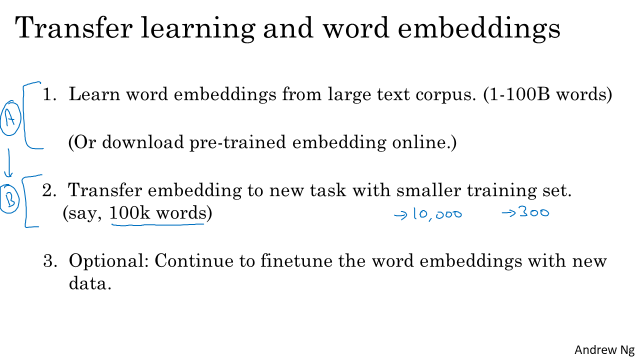
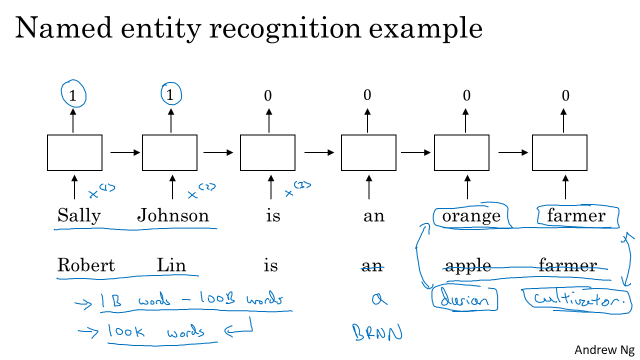
2.1 Word representation



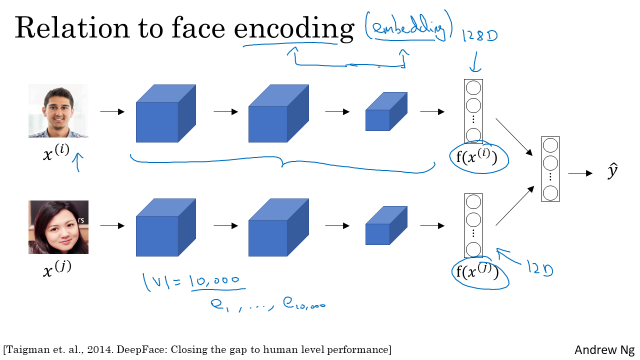


300 dimensional data 🡪 2D

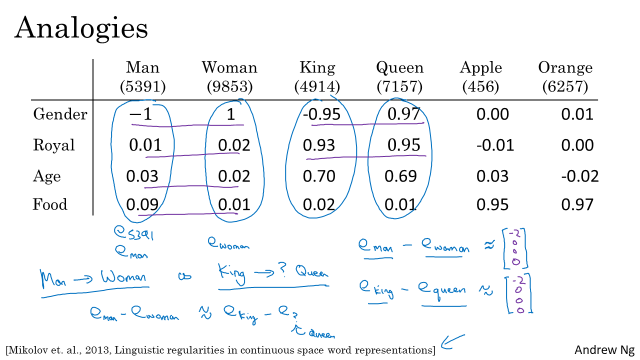
**2.2 Using Word Embeddings**



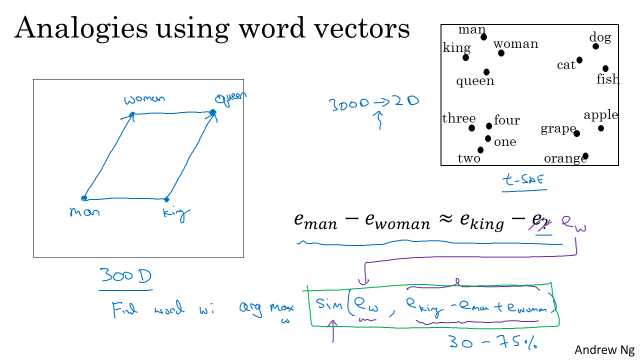
And then, finally, as you train your model on your new task, on your named entity recognition task with a smaller label data set, one thing you can optionally do is to continue to fine tune, continue to adjust the word embeddings with the new data. In practice, you would do this only if this task 2 has 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.

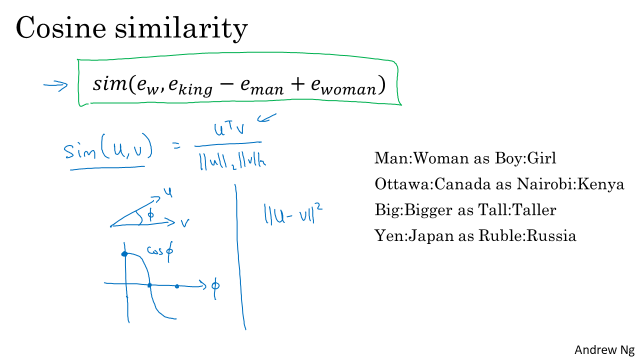


**2.3 Properties of Word Embeddings**



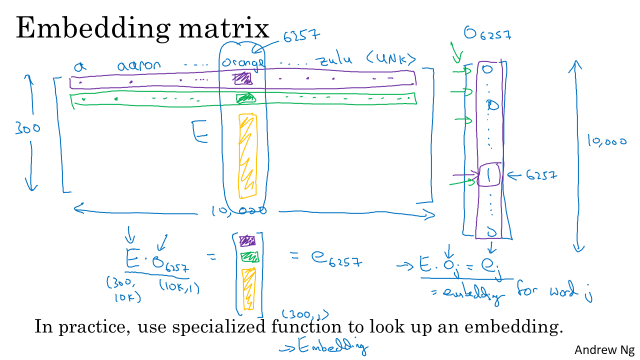
Here we are using 4 dimensional embedding.





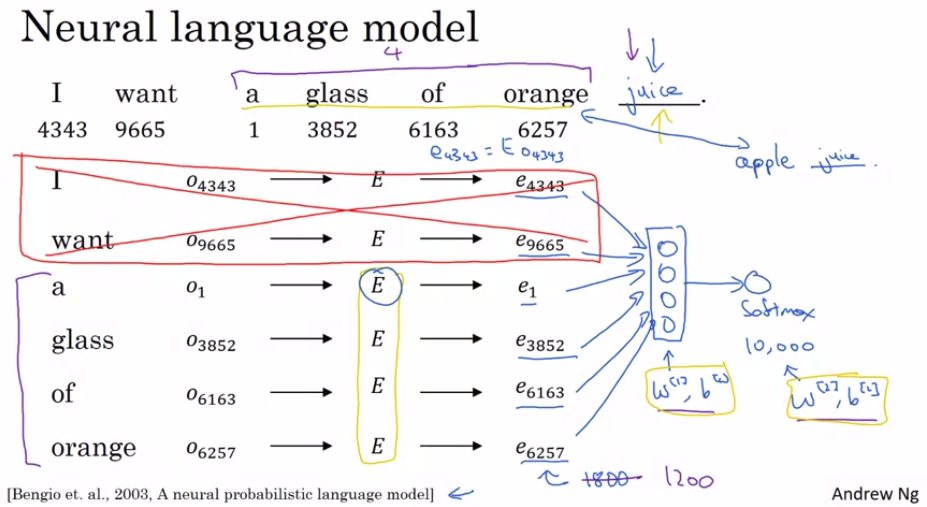
If you want, you can also use square distance or Euclidian distance, u-v squared. Technically, this would be a measure of dissimilarity rather than a measure of similarity. So we need to take the negative of this, and this will work okay as well. Although I see cosine similarity being used a bit more often.

**2.4 Embedding Matrix**

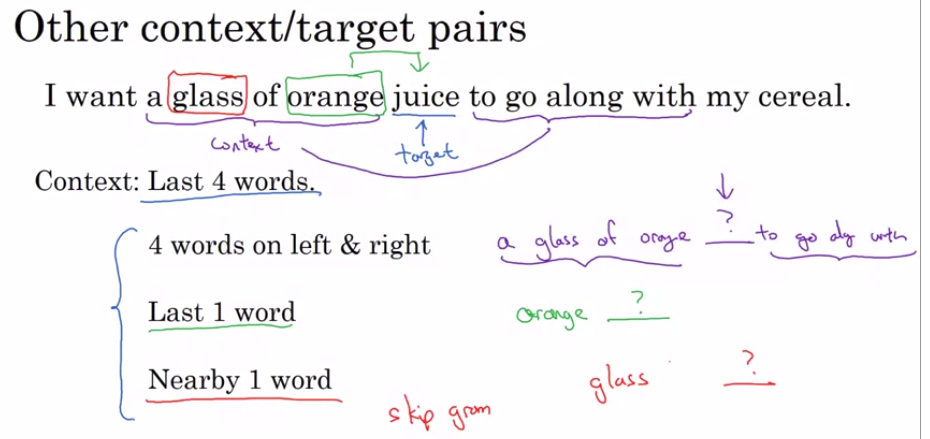


So, it's actually not efficient to use a matrix vector multiplication to implement this because if we multiply a whole bunch of things by zeros and so the practice, you would actually use a specialized function to just look up a column of the Matrix E rather than do this with the matrix multiplication. But writing of the map, it is just convenient to write it out this way. So, in Cara's 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.

**2.5 Learning word embeddings**



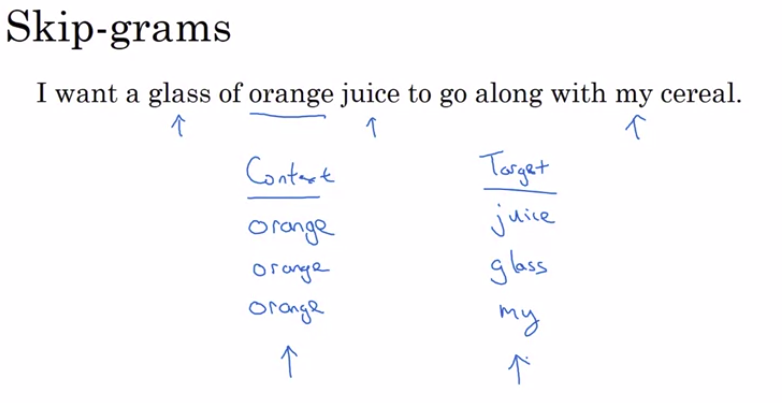
And so, if you're always using a four word history, this means that your neural network will input a 1,200 dimensional feature vector, go into this layer, then have a softmax and try to predict the output. And again, variety of choices. And using a fixed history, just means that you can deal with even arbitrarily long sentences because the input sizes are always fixed.



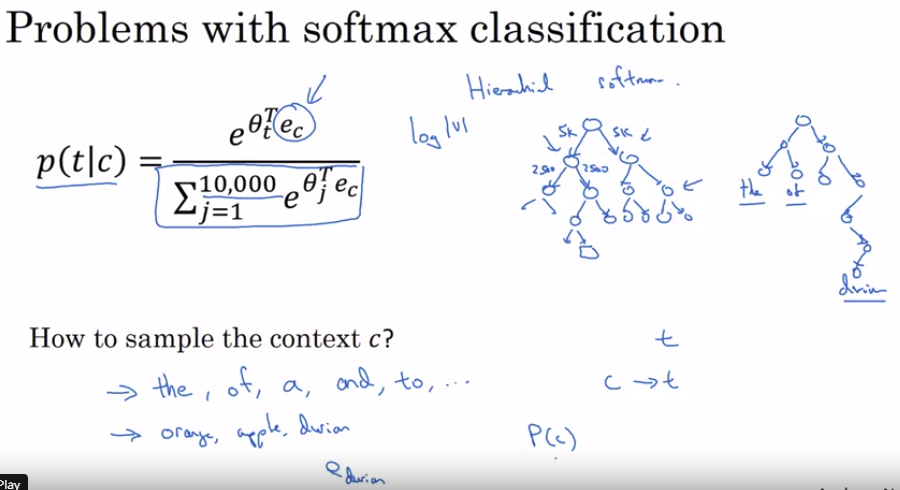
So what researchers found was that if you really want to build a language model, it's natural to use the last few words as a context. But if your main goal is really to learn a word embedding, then you can use all of these other contexts and they will result in very meaningful work embeddings as well.

**2.6 Word2Vec**

Given a set of sentences (also called **corpus**) the model loops on the words of each sentence and either tries to use the current word of to predict its neighbors (its context), in which case the method is called “Skip-Gram”, or it uses each of these contexts to predict the current word, in which case the method is called “Continuous Bag Of Words” (CBOW). The limit on the number of words in each context is determined by a parameter called “**window size**”.



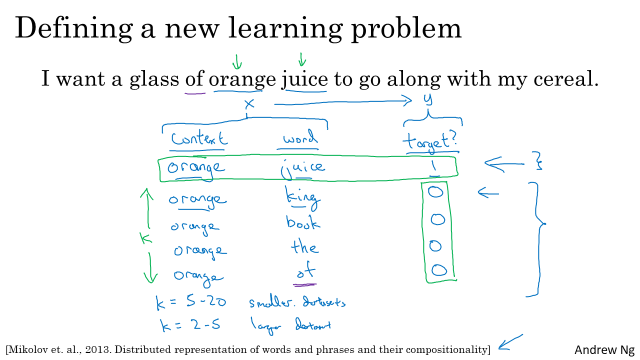




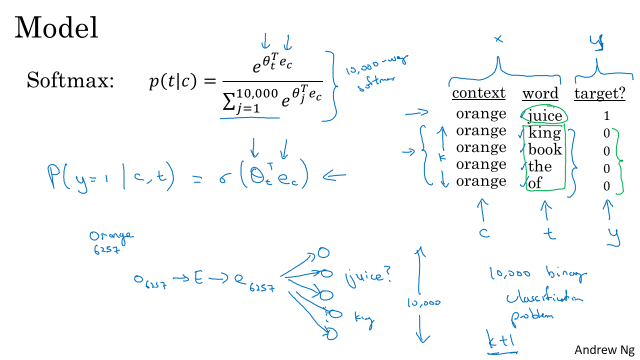
Now, it turns out there are a couple problems with using this algorithm. And the **primary problem is computational speed**. In particular, for the softmax model, every time you want to evaluate this probability, you need to carry out a sum over all 10,000 words in your vocabulary. And maybe 10,000 isn't 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 binary 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**.

**2.7 Negative Sampling**



K=5~20 smaller datasets, K=2~5 for large datasets, In this example, k=4



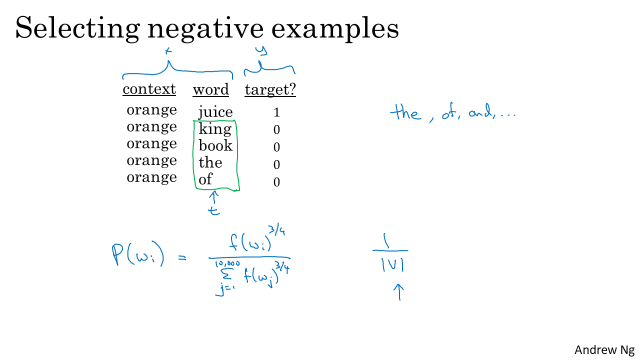
K to 1 ratio of positive to negative samples

10,000 binary logistic classification problem

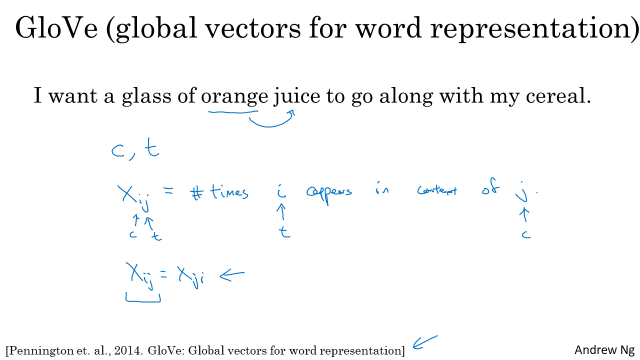
So instead of having one giant 10,000 way Softmax, which is very expensive to compute, we've instead turned it into 10,000 binary classification problems, each of which is quite cheap to compute. And on every iteration, we're only going to train five of them or more generally, k + 1 of them, of k negative 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.

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?

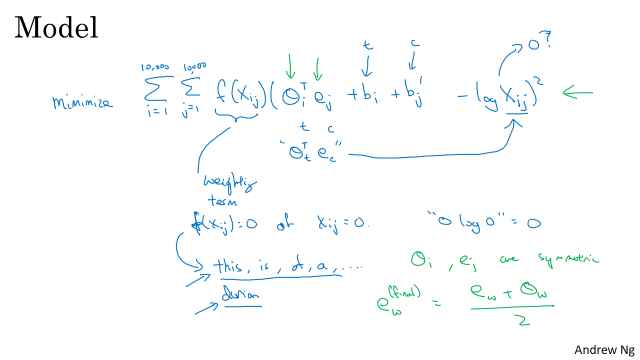
What they did was they sampled proportional to their frequency of a word to the power of three-fourths. So if f of wi is the observed frequency of a particular word in the English language or in your training set corpus, then by taking it to the power of three-fourths, this is somewhere in-between the extreme of taking uniform distribution. And the other extreme of just taking whatever was the observed distribution in your training set.



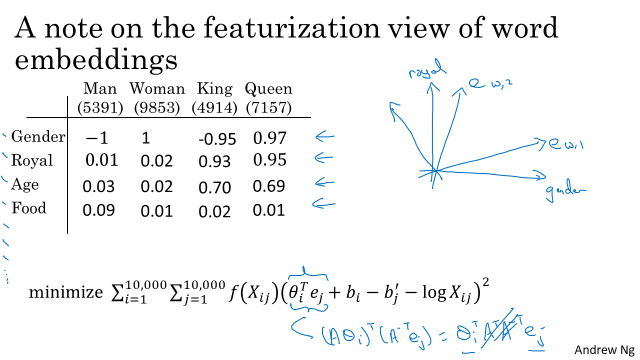
**2.6 GloVe word vectors**



is a count that captures how often do words i and j appear with each other, or close to each other.

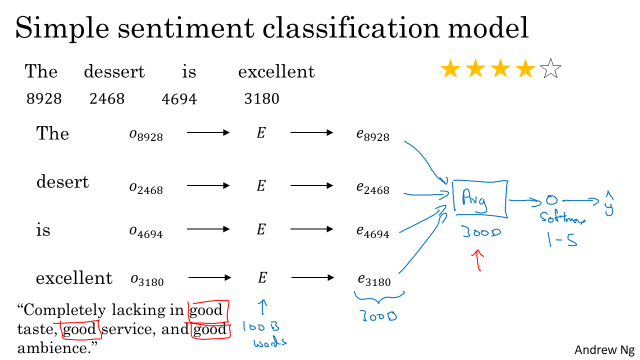


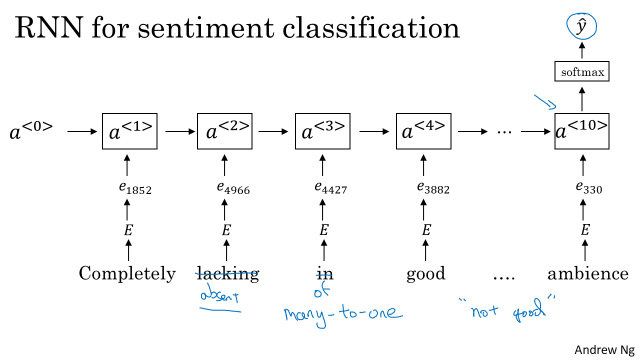
One funny thing about this algorithm is that the roles of theta and e are now completely symmetric. So, and are symmetric in that, if you look at the math, they play pretty much the same role and you could reverse them or sort them around, and they actually end up with the same optimization objective.



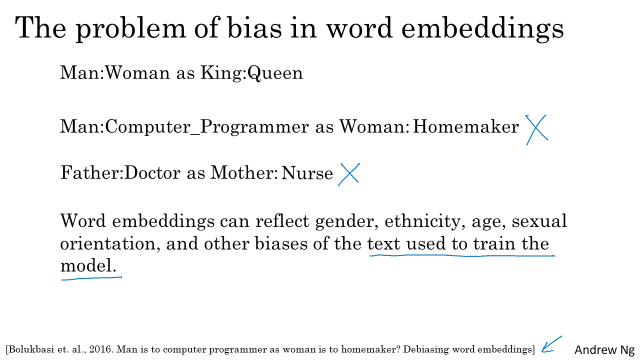
With an algorithm like this, you can't guarantee that the axis used to represent the features will be well-aligned with what might be easily humanly interpretable axis. In particular, the first feature might be a combination of gender, and royal, and age, and food, and cost, and size, is it a noun or an action verb, and all the other features. It's very difficult to look at individual components, individual rows of the embedding matrix and assign the human interpretation to that. But despite this type of linear transformation, the parallelogram map that we worked out when we were describing analogies, that still works. And so, despite this potentially arbitrary linear transformation of the features, you end up learning the parallelogram map for figure analogies still works. So, that's it for learning word embeddings.

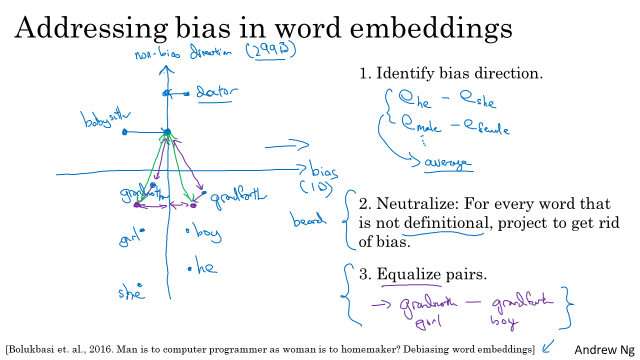
**2.7 Sentiment Classification**





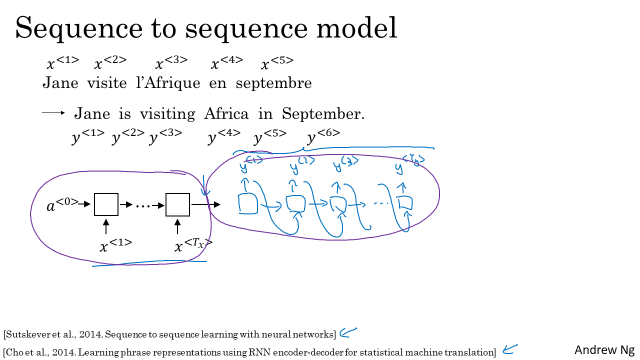
**2.8 Debiasing Word Embeddings**

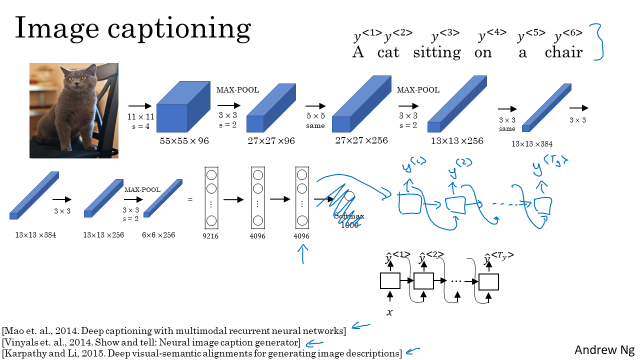




**Week 3**

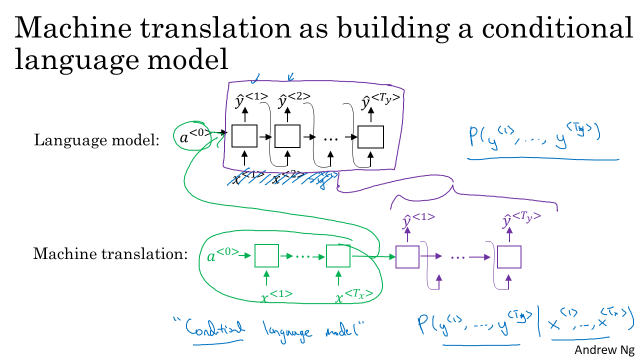
**3.1 Basic Models**

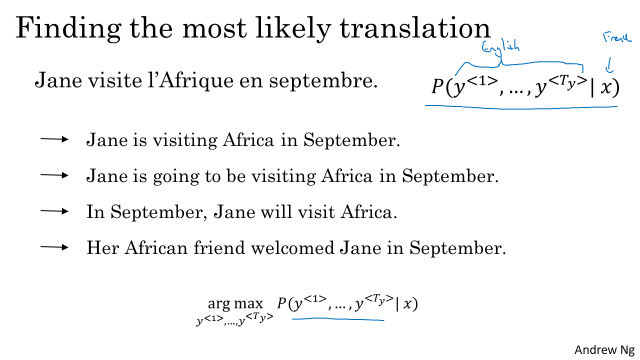
Encoder network 🡪 Decoder network

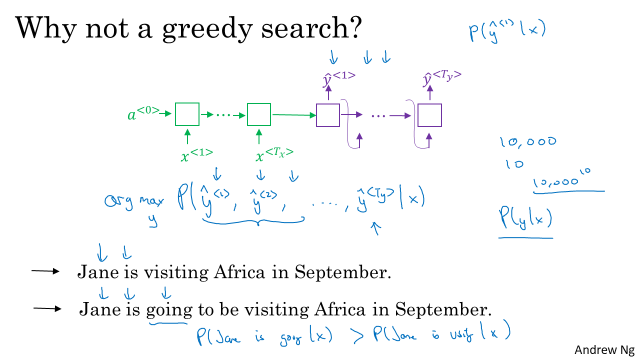


So, this is actually the AlexNet architecture and if we get rid of this final Softmax unit, the pre-trained AlexNet can give you a 4096-dimensional feature vector of which to represent this picture of a cat. And so this pre-trained network can be the encoder network for the image and you now have a 4096-dimensional 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.

**3.2 Picking the most likely sentences**



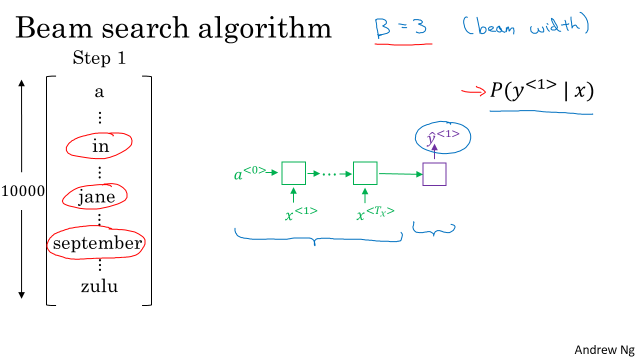


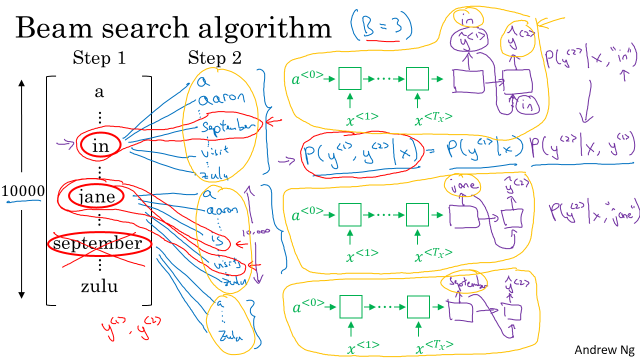


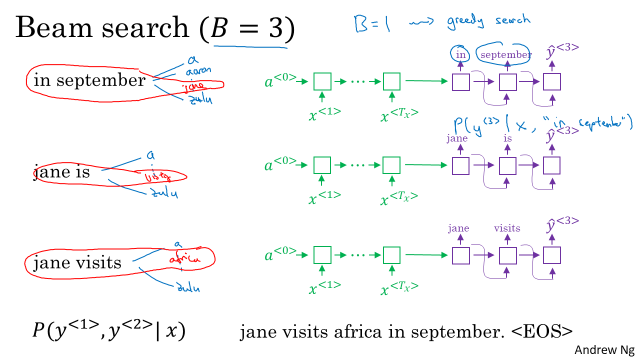
What is greedy search? Well, greedy search is an algorithm from computer science which says to generate the first word just pick whatever is the most likely first word according to your conditional language model. Going to your machine translation model and then after having picked the first word, you 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

3.3 Beam Search

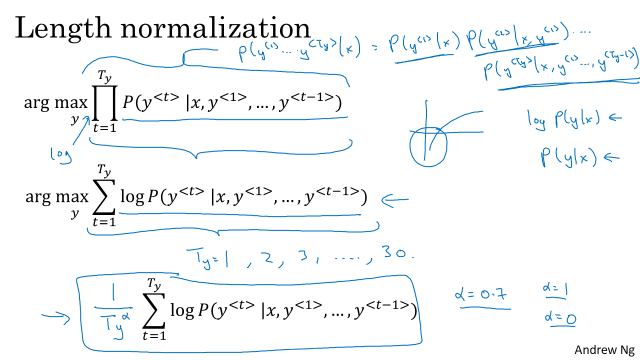


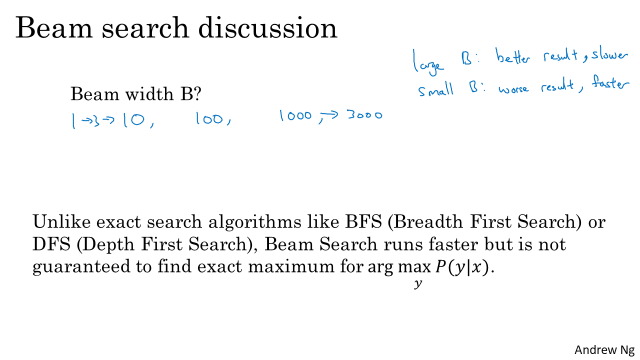




So with a beam of 3 beam searched considers three possibilities at a time. Notice that if the beam width was said to be equal to one, say cause there's only one, then this essentially becomes the greedy search algorithm which we had discussed in the last video but by considering multiple possibilities 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.

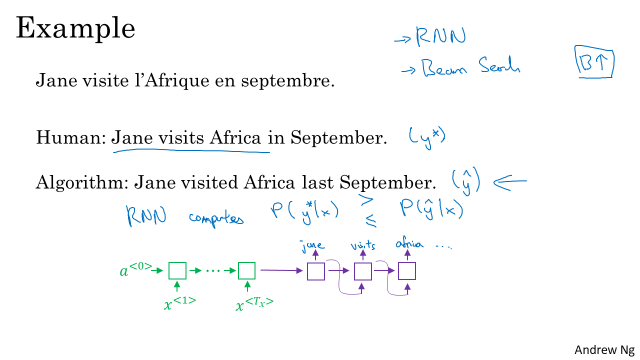
**3.4 Refinement to Beam search**

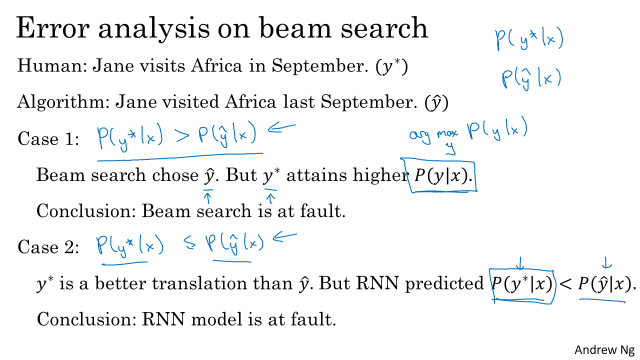


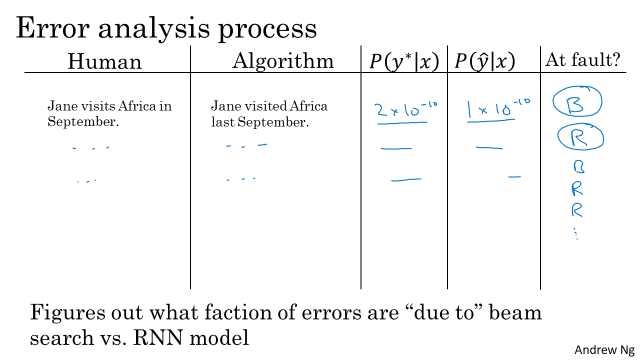


Whereas if you use a very small beam width, then you get a worse result because you're just keeping less possibilities in mind as the algorithm is running. But you get a result faster and the memory requirements will also be lower. So in the previous video, we used in our running example a beam width 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.

**3.5 Error analysis in beam search**

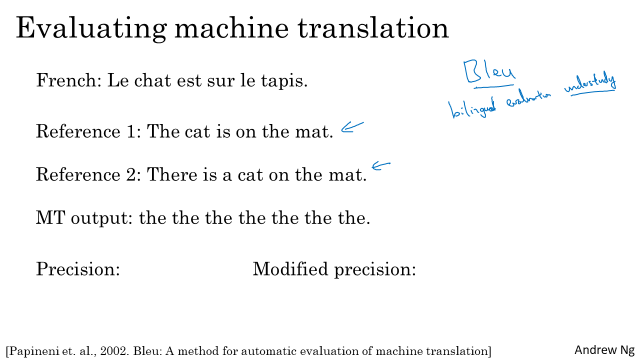






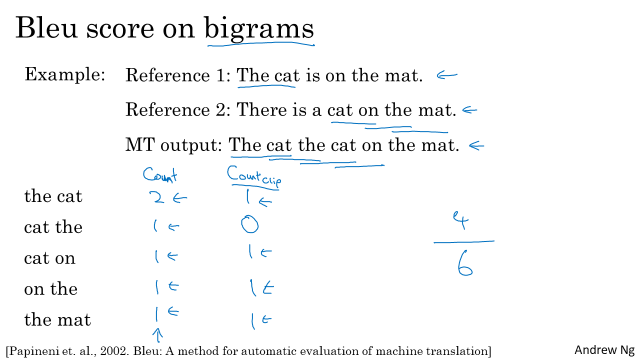
Through this, you can try to figure out which of these two components is responsible for more errors. And only if you find that beam search is responsible for a lot of errors, then maybe is we're working hard to increase the beam width. Whereas in contrast, if you find that the RNN model is at fault, 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

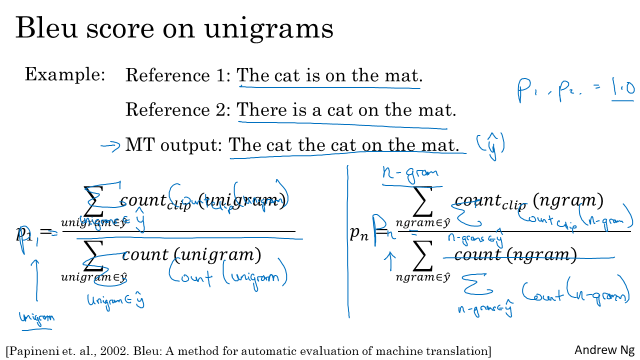
**3.6 Bleu Score**

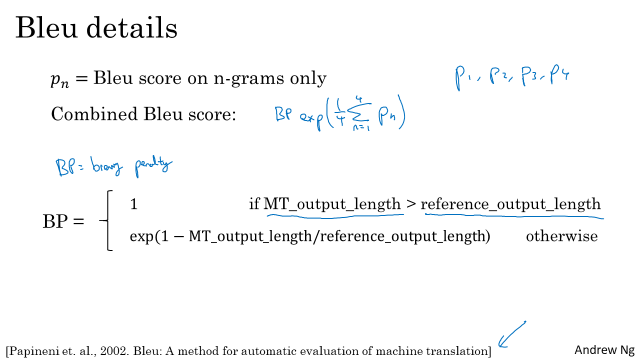


What the BLEU score does is given a machine generated translation, it allows you to automatically compute a score that measures how good is that machine translation.

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, by the way, stands for bilingual evaluation,



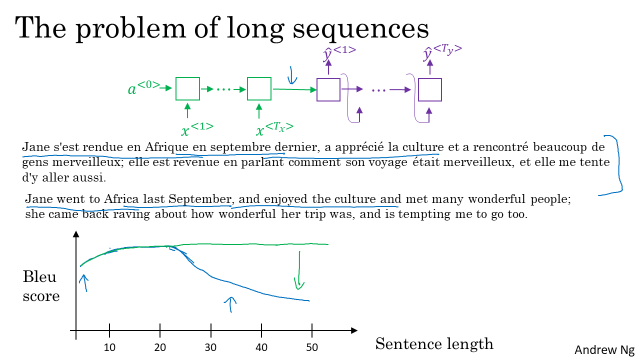




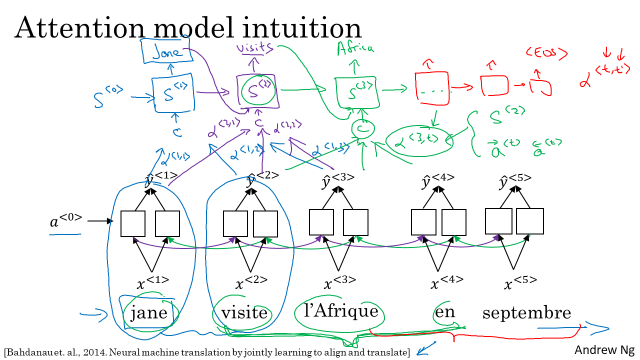
BP penalty. So BP, Stands for brevity penalty. The details maybe aren't super important. But to just give you a sense, it turns out that if you output very short translations, it's easier to get high precision. Because probably most of the words you output appear in the references.

But we don't want translations that are very short. So the BP, or the brevity penalty, is an adjustment factor that penalizes translation systems that output translations that are too short. So the formula for the brevity penalty is the following. It's equal to 1 if your machine translation system actually outputs things that are longer than the human generated reference outputs. And otherwise is some formula like that that overall penalizes shorter translations.

**3.7 Attention Model Intuition**



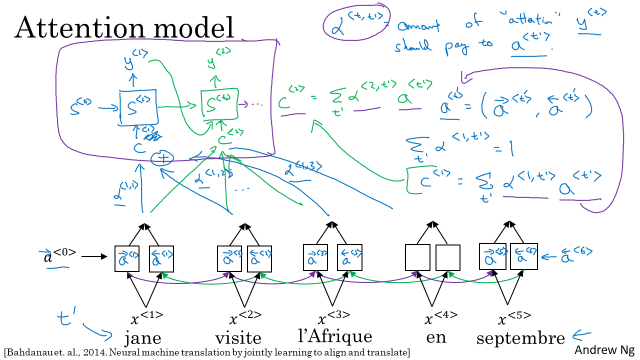
What you see for the Encoder-Decoder architecture above is that, it works quite well for short sentences, so we might achieve a relatively high Bleu score, but for very long sentences, maybe longer than 30 or 40 words, the performance comes down. The Bleu score might look like this as the sentence that 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.

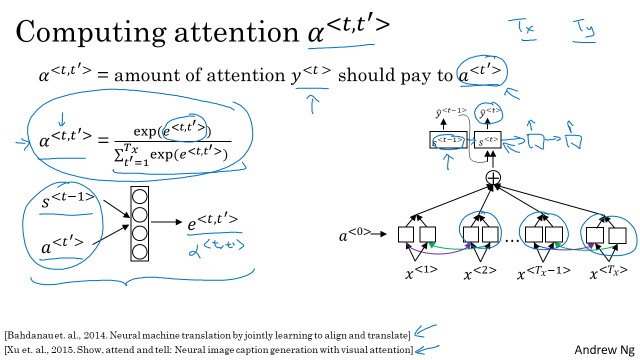


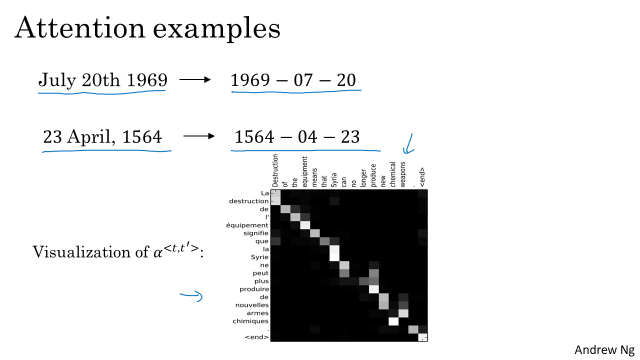
What the Attention Model would be computing is a set of attention weights and we're going to use Alpha one, one to denote when you're generating the first words, how much should you be paying attention to this first piece of information here. And then we'll also come up with a second that's called Attention 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 denote 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.

**3.8 Attention Model**

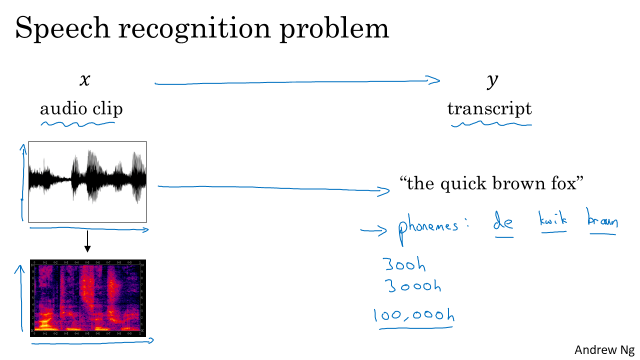


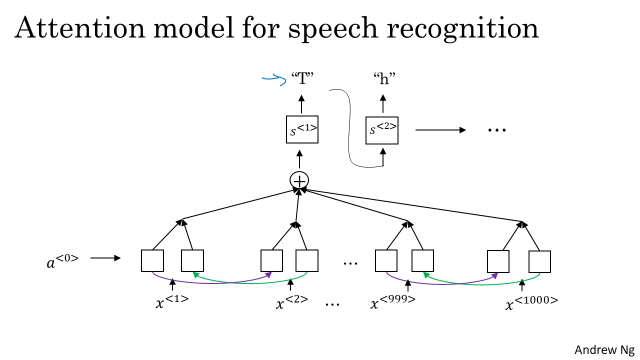




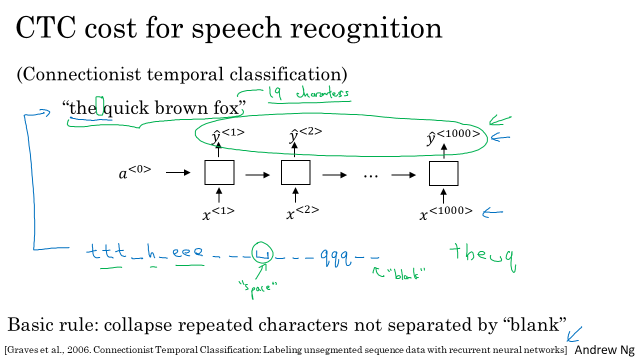
The corresponding input and output words you find that the attention waits will tend to be high. Thus, suggesting that when it's generating a specific word in output is, usually paying attention to the correct words in the input and all this including learning where to pay attention when was all learned using propagation with an attention model.

**3.9 Speech recognition**

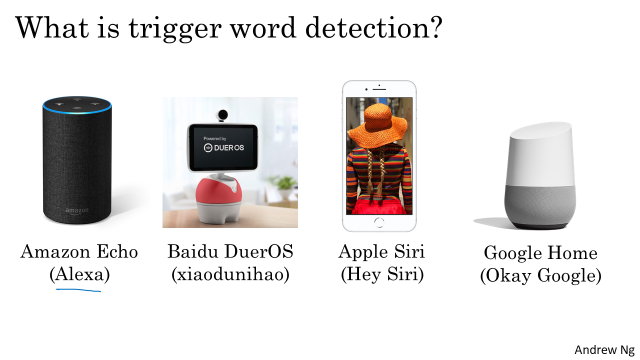


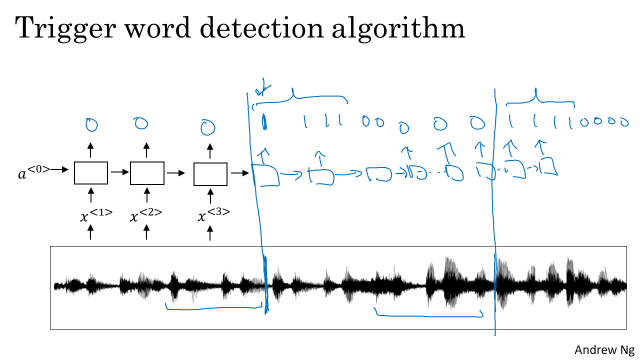


So, how do you build a speech recognition system? In the last video, we're talking about the attention model. So, one thing you could do is actually do that, where on the horizontal axis, you take in different time frames of the audio input, and then you have an attention model try to output the transcript 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



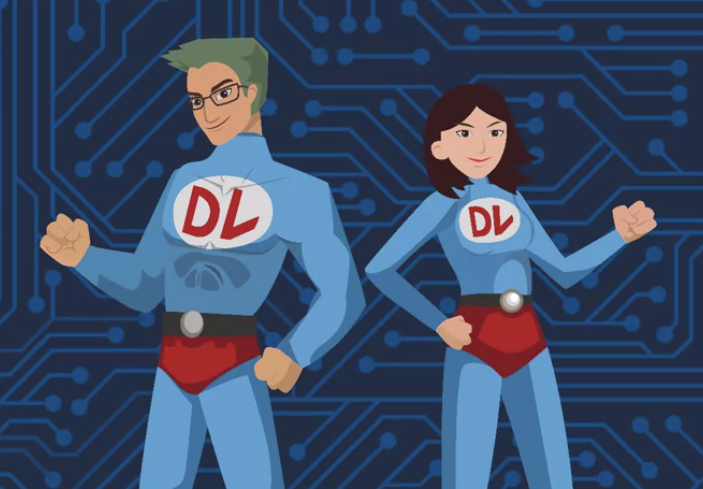
**3.10 Trigger Word Detection**





**3.11 Summary and Thank You**

Deep learning is super power.



**Week 4**

**4.1 Transformer Network Intuition**

Diagram

Description automatically generated

RNN 🡪 GRU, LSTM (To conquer the vanishing gradient which made it hard to capture long range dependencies and sequences)

As we move from our RNNs to GRU to LSTM ,the models became more complex. And all of these models are **still sequential** models in that they i**ngested** the input, maybe the input sentence **one word or one token at the time**. And so, as as if each unit was like a bottleneck to the flow of information. Because to compute the output of this final unit, for example, you first have to compute the outputs of all of the units that come before.

While for the transformer architecture, which allows you to run a lot more of these computations for an entire sequence in parallel. So you can ingest an entire sentence all at the same time, rather than just processing it one word at a time from left to right.

The major innovation of the transformer architecture is combining the use of attention based representations and a CNN convolutional neural network style of processing

Diagram

Description automatically generated

**4.2 Self Attention**

Graphical user interface, diagram, application

Description automatically generatedText

Description automatically generated with medium confidence

Diagram

Description automatically generated

**Q = interesting questions about the words in a sentence,**

**K = qualities of words given a Q,**

**V = specific representations of words given a Q Scaled dot-product attention**

**Q:** Vector(Linear layer output) related with what we encode(output, it can be output of encoder layer or decoder layer)  
**K:**Vector(Linear layer output) related with what we use as input to output.  
**V**: Learned vector(Linear layer output) as a result of calculations, related with input

1- **Encoder Self attention**  
Q = K = V = Our source sentence(English)

2- **Decoder Self attention**  
Q = K = V = Our target sentence(German)

3- **Decoder-Encoder attention**  
Q = Our target sentence(German)  
K = V = Our source sentence(English)

attention = Softmax( (**Q \* K**) / Scale)  
enbedding for attention layer = attention \* **V**

The key/value/query concept is analogous to retrieval systems. For example, when you search for videos on Youtube, the search engine will map your **query** (text in the search bar) against a set of **keys** (video title, description, etc.) associated with candidate videos in their database, then present you the best matched videos (**values**).

To recap, associated with each of the five words you end up with a query, a key, and a value. **The query lets you ask a question about that word**, such as what's happening in Africa. The **key looks at all of the other words, and by the similarity to the query**, helps you figure out **which words gives the most relevant answer to that question**. In this case, visite is what's happening in Africa, someone's visiting Africa. Then finally, **the value allows the representation to plug in how visite should be represented within A^3**, within the representation of Africa. This allows you to come up with a representation for the word Africa that says this is Africa and someone is visiting Africa. This is a much more nuanced, much richer representation for the world than if you just had to pull up the same fixed word embedding

for every single word without being able to adapt it based on what words are to the left and to the right of that word. We've all got to take into account and in the context. Now, you have learned about the self-attention mechanism. We're going to put a big four-loop over this whole thing and that will be the multi-headed attention mechanism.

**4.3 Multi-head attention**

Multi headed attention is basic A for loop over the self attention process. So you end up with multiple versions of these representations. And it turns out that these representations, which will be very rich representations, can be used for machine translation or other NLP toss to create effectiveness

**Diagram

Description automatically generated**

**4.4 Transformer Network**

Diagram

Description automatically generated

What masking does is it blocks out the last part of the sentence to mimic what the network will need to do at test time or during prediction. In other words, all that mask multi- head attention does is

repeatedly pretends that the network had perfectly translated. Say the first few words and hides the remaining words to see if given a perfect first part of the translation, whether the neural network can predict the next word in the sequence accurately.