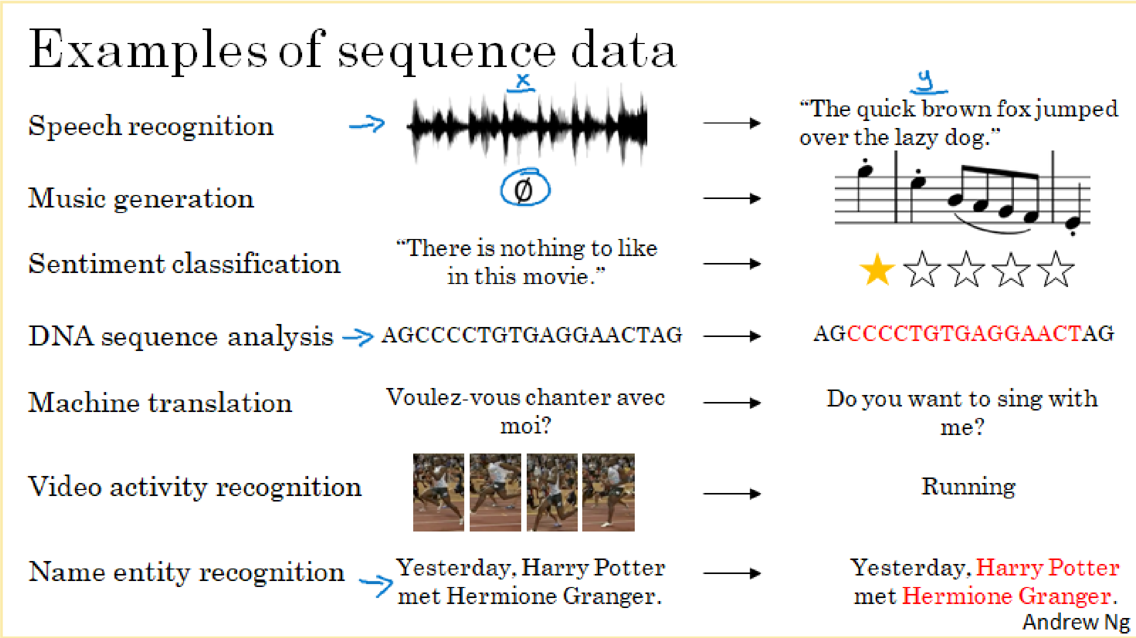
Course 5: Sequence Models

Content

1. Recurrent Neural Networks

1.1. Why sequence models?

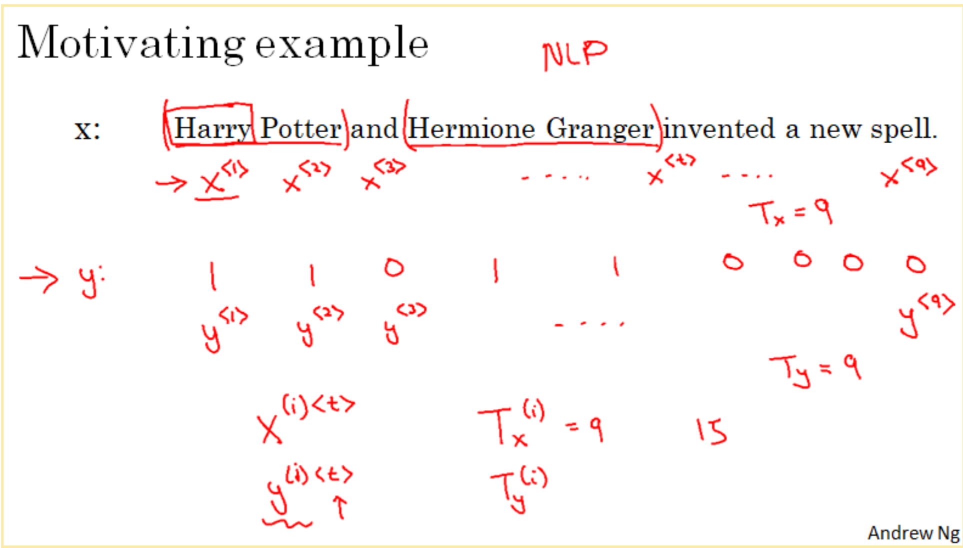
Models like recurrent neural networks or RNNs have transformed speech recognition, natural language processing and other areas. Some examples of the use of sequence data are:



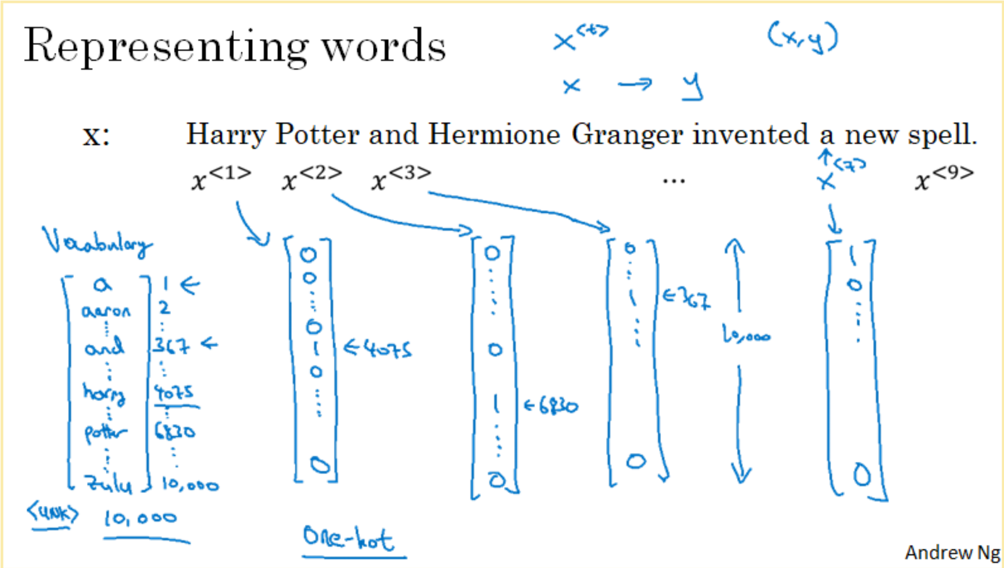
1.2. Notation

Let's say we want a sequence model to know where the people's names in the sentence are. Let's suppose our output is the y vector, which takes 1 or 0 values depending whether a word is a name or not.

Our feature vector X is the sentence. Each word is denoted by $x^{<t>}$ and T_x or T_y is the length of the vectors. Also, if we want to note that a feature vector corresponds to the i-th observation, we will use $x^{(i)<t>}$.

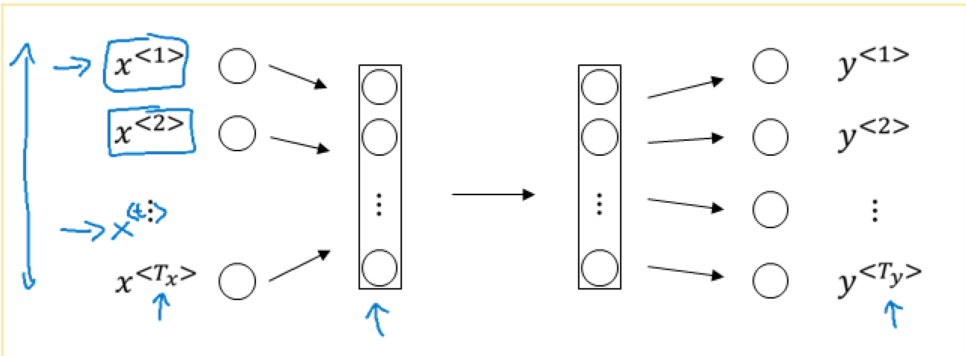


In addition, if we are in an NLP problem, one way to represent each individual word is with one-hot encoding (by using a vocabulary vector):



1.3. Recurrent Neural Network Model

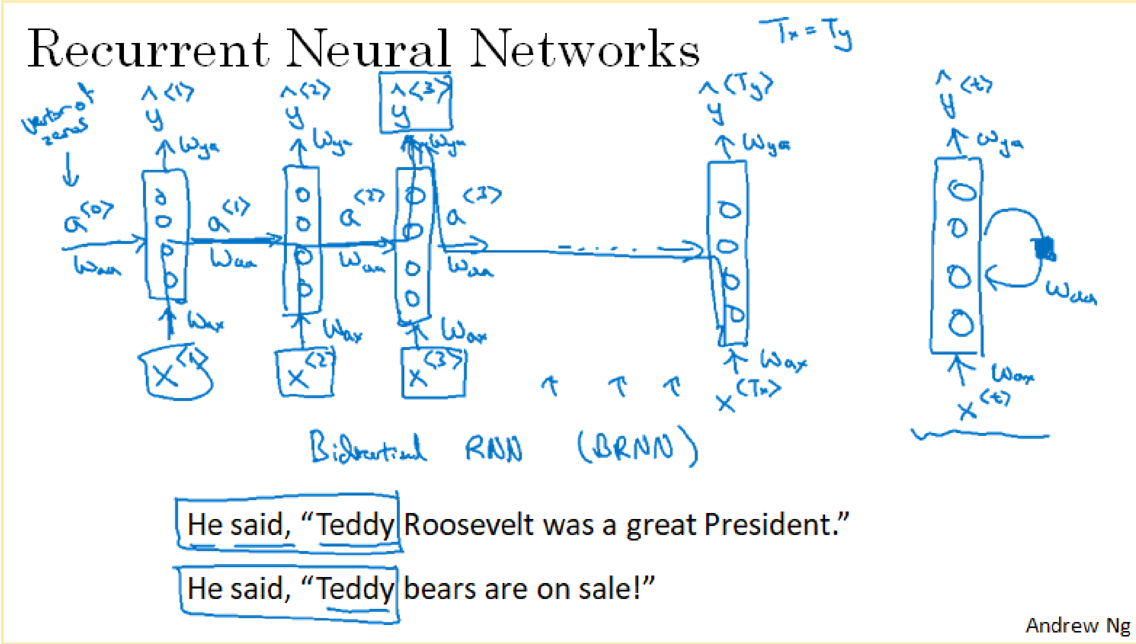
With the example we have seen, we could try to use a standard network to learn the patterns and map X to y:



But this turns out to not work well, and there are two main problems that explain this:

- The inputs and outputs can be different lengths in different examples.
- Doesn't share features learned across different positions of text. This is similar to what happened in CNN: we wanted things learned for one part of the image to generalize quickly to other parts of the image. A similar idea applies to sequence data.

Recurrent Neural Networks don't have any of these problems. Let's build one up:

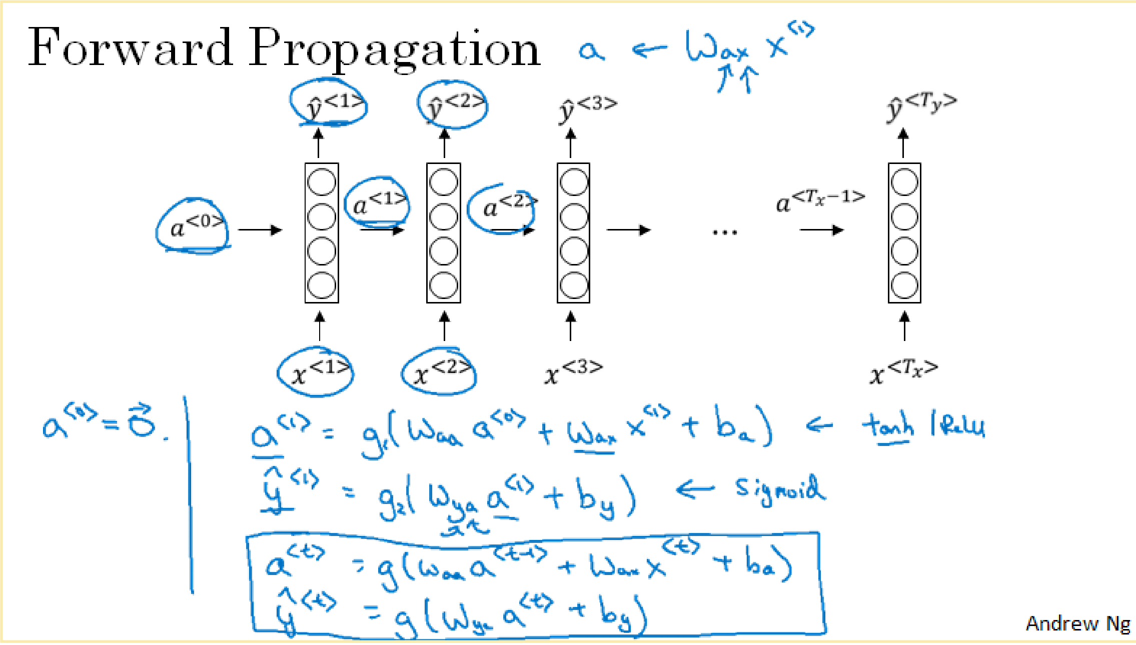


We'll pass the first word of the example to the first layer and then output a prediction. Then, we'll feed the second word into the second layer but we'll also use the activation of the first layer (this way we are using the "information" of the first layer) to get the output. And we'll continue doing this.

We have some different sets of parameters: Waa, Wya, Wax (which we'll see later). In addition, we need to initialize the first activation. We'll usually initialize it with zeros or randomly.

It is important to note that one weakness of this architecture is that for every word, we only use the information **before** it. We can see the effects of this in the two sentences in the above slide. To know whether Teddy is a person's name or not, we would only use the information of the words "*He said*". We will address this issue later with Bidirectional RNN or BRNNs.

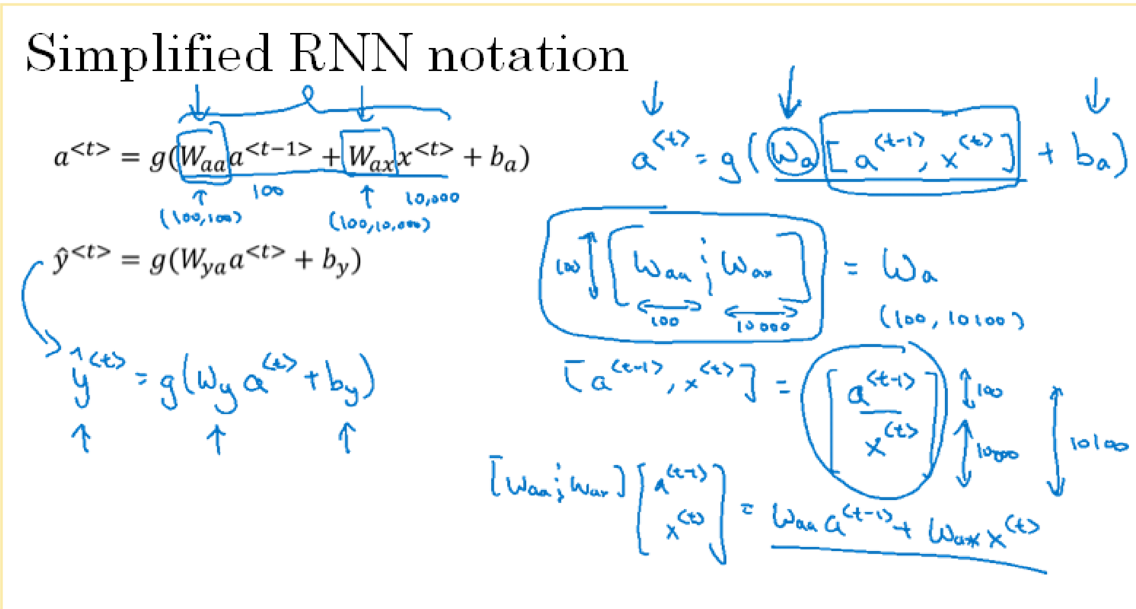
Let's write explicitly the calculations done:



When using the subscripts for the weights, the first one denotes the weight will be used to compute an alike quantity, and the second one is what we will be using for computing it.

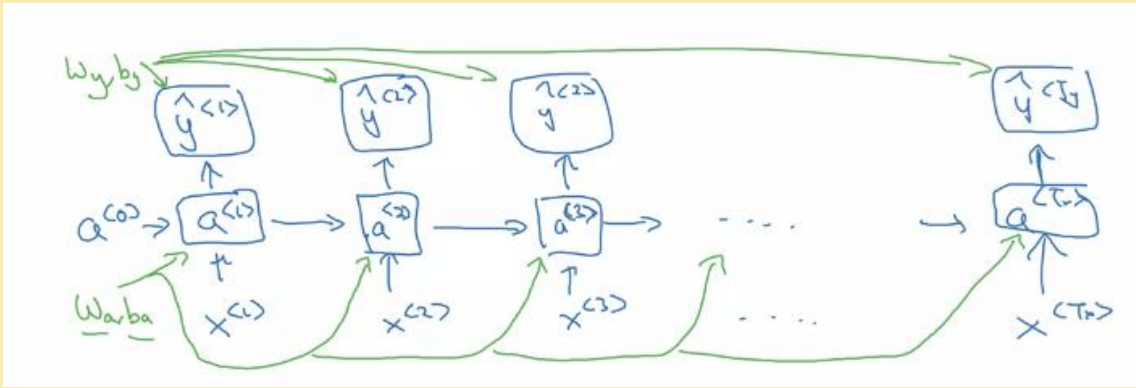
The *g1* activation functions can be *tanh* and *ReLu* (being the first one the most used). The *g2* activations can be different ones like *sigmoid*.

So, the squared equations are the ones we'll use to calculate the activations and outputs of the network. However, we are going to simplify this notation:



1.4. Backpropagation through time

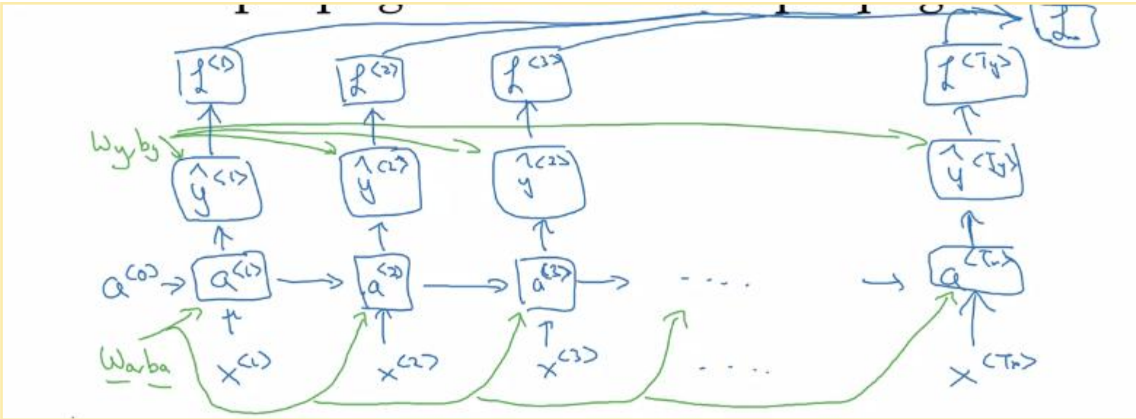
We have already seen the process of forward propagation:



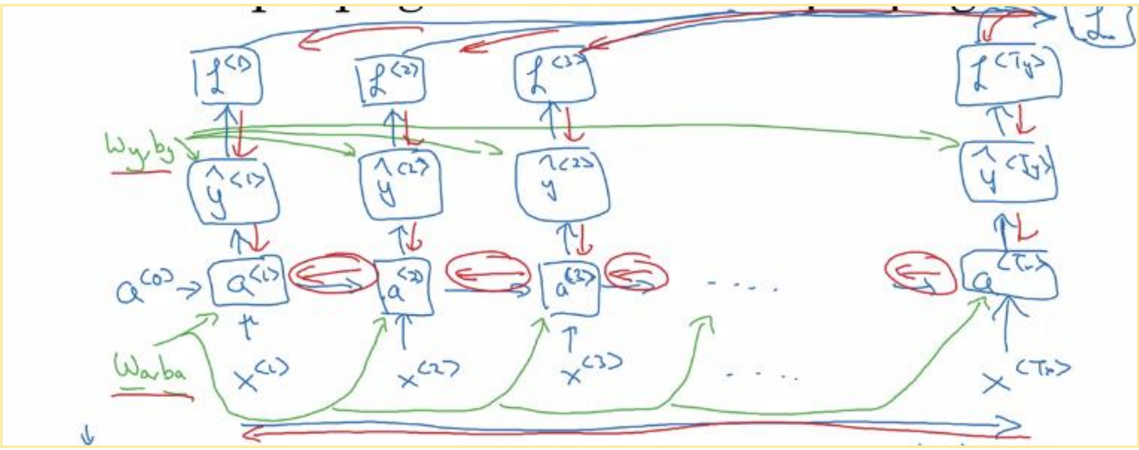
Now, to compute backpropagation, we need a loss function. We will use the binary cross-entropy in this example:

$$\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T_y} \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

So we have this:



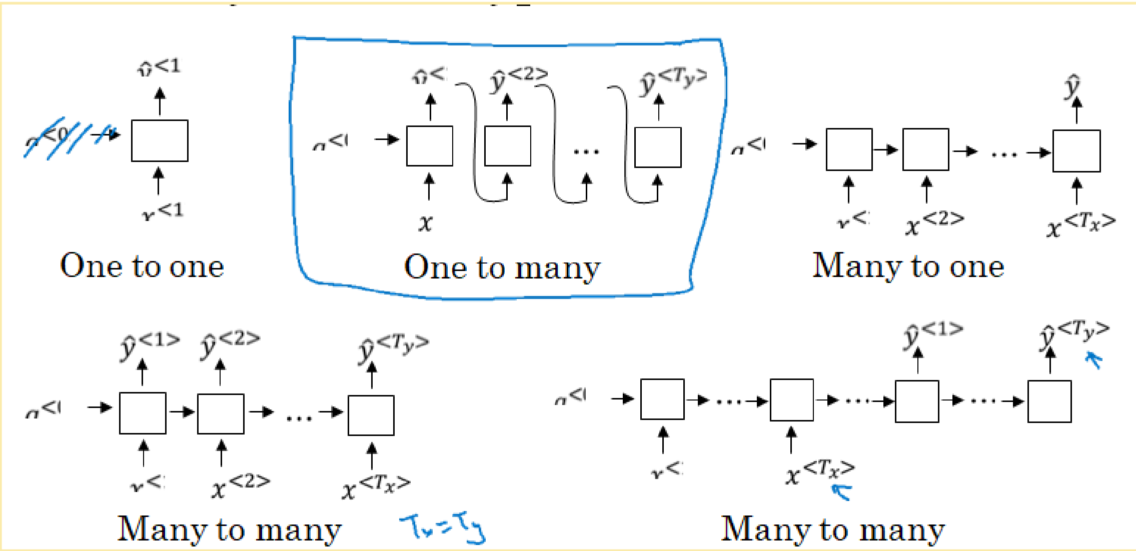
Then, to backpropagate we will follow the red arrows and perform a **backpropagation through time**:



Until now, we have seen an example in which the length of the input is the same as the length of the output. Let's cover a wide range of RNN architectures:

1.5. Different types of RNN

We can have different types of architectures depending on the definition o the problem:



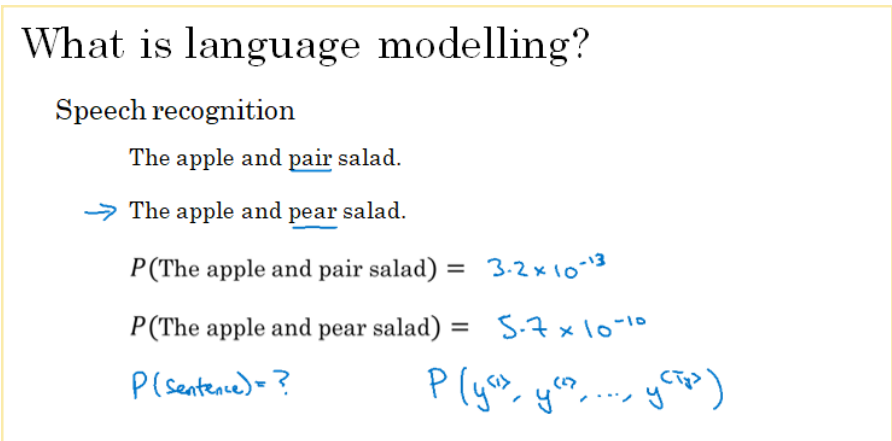
Some examples for these architectures are:

- One to one: a simple neural network
- One to many: music generation
- Many to one: sentiment classification
- Many to many: named entity recognition, machine translation

There are some subtleties that we should see when talking about the one to many architecture, which can be used for a sequence generation problem.

1.6. Language model and sequence generation

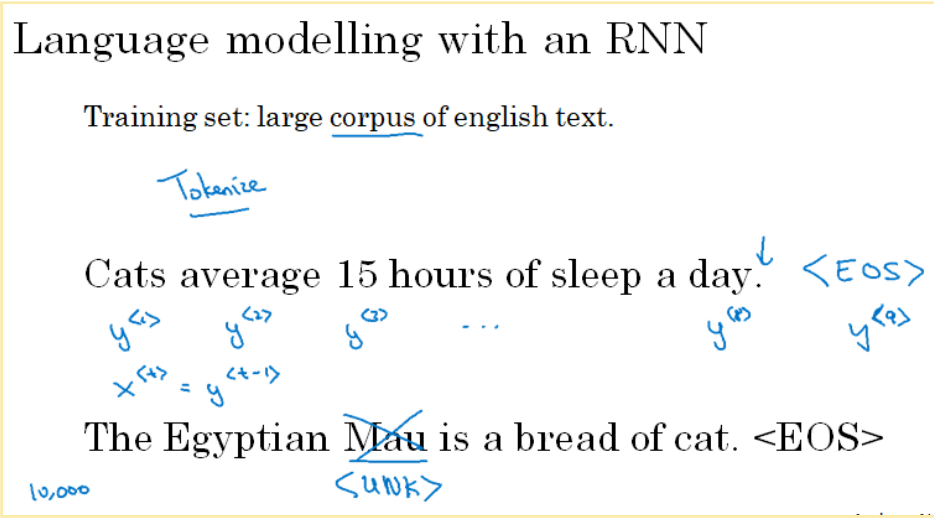
Imagine if we had a speech recognition application that wants to know which one of the two options below is the best one:



The two sentences sound exactly the same. If we could calculate the probability of each of the two sentences, we could choose which one is most accurate.

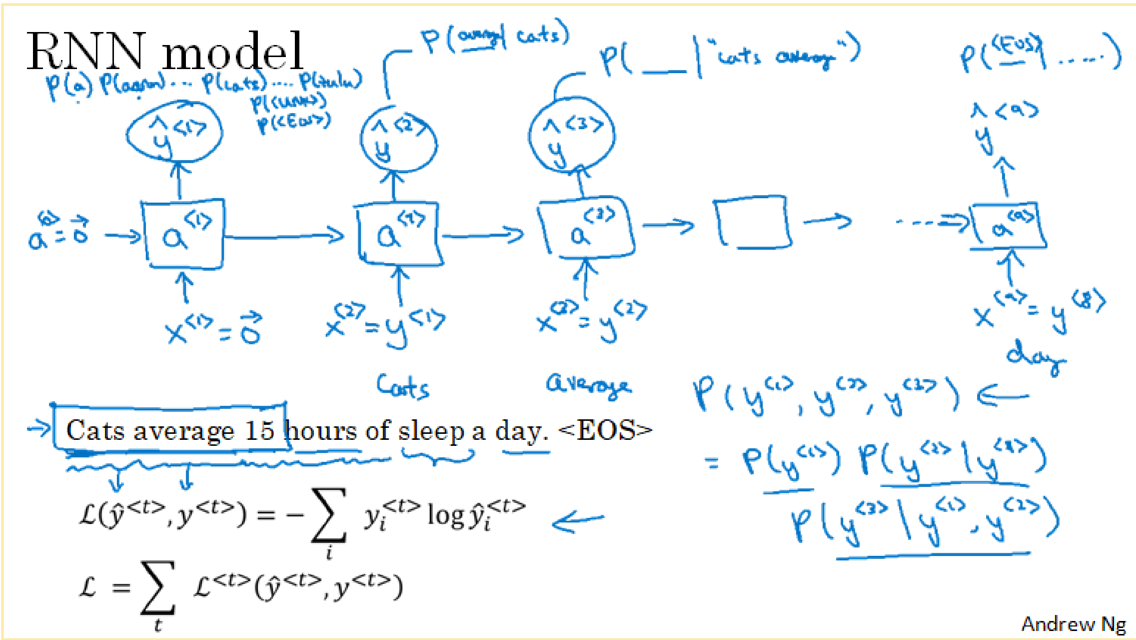
So that's what a language model does: we input a sentence and it estimates a probability of that particular sequence of words. So, how do we build a language model with an RNN?

First of all, we need a training set: a large corpus of English (or whatever language) text. The second thing we need to do is tokenize the sentences, as it is shown in the slide:



We can also use the <EOS> token for denoting the end of a sentence and the <UNK> token for the words that are not present in the training set.

Then, we'll feed the sentence into the RNN as it is shown below:



In the first step, the output of the layer will be a vector in which each element is the probability of the i-th word in the vocabulary to be the first word in the sentence.

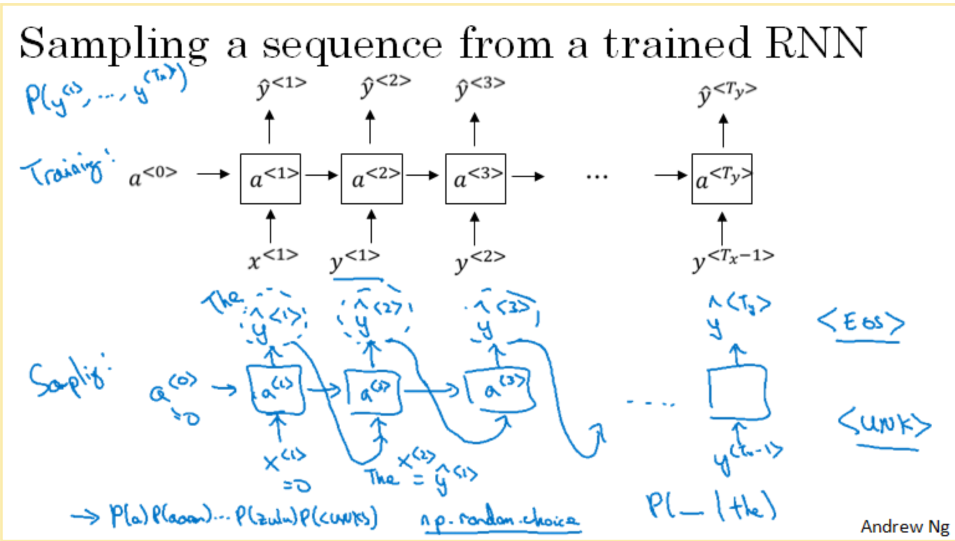
Then, the second's unit output will be a similar vector but taking into account which was the first word. This means, the i-th element of the vector will represent the probability of the i-th word in the vocabulary to be after the word of the previous step.

So if we go step by step, we will be able to get the most probable word to go after a certain sequence on each step.

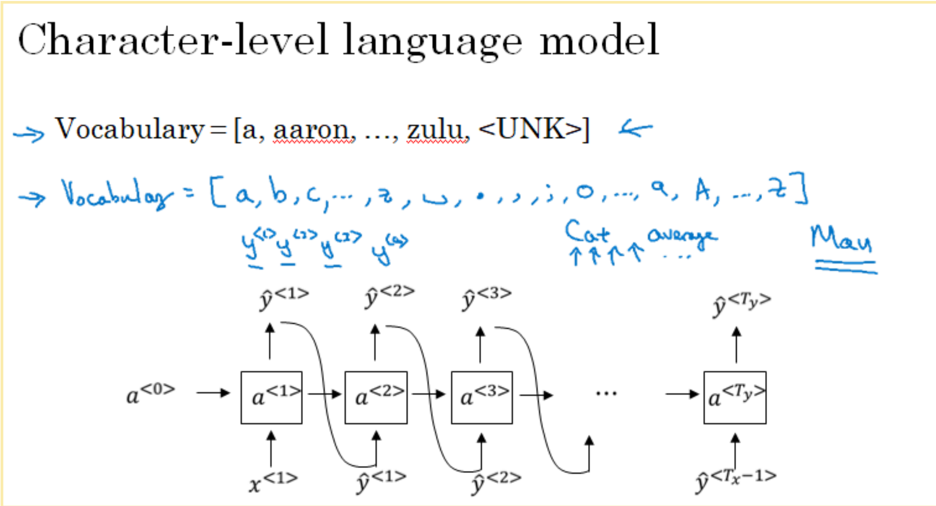
Once we have trained a sequence model, one of the ways we can informally get a sense of what it has learned is to have a sample novel sequence.

1.7. Sampling model sequences

What we are going to do is set the first word of a sentence and then "run" the RNN. We'll do it by taking a random value according to the distribution of each output on each layer. So in the first step, the RNN will output a second word that follows the first one we set, and then the second layer will output a word taking into account the two previous words. And so on:

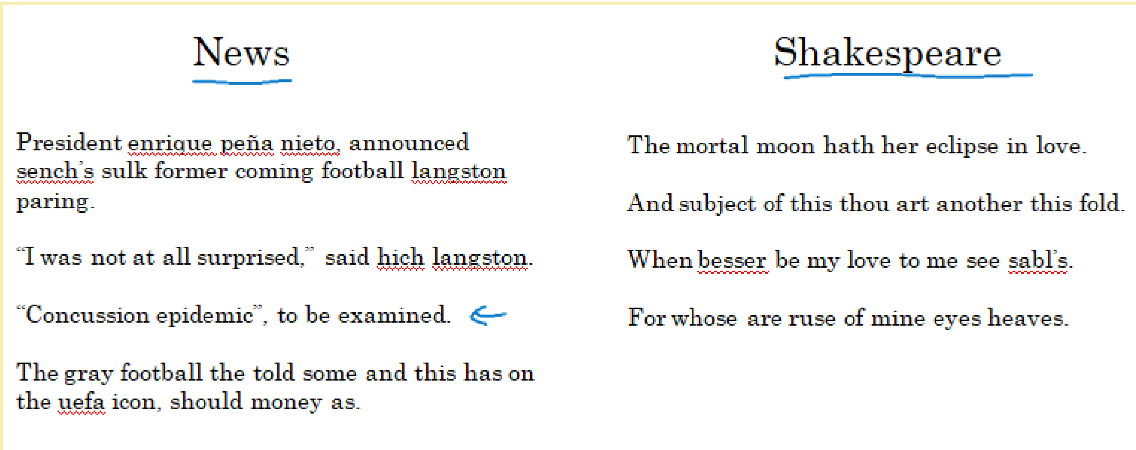


We could also build a character-level language model:



However, they are really computationally expensive, so they are not really extensively used yet.

We can see some examples of sequence generation:



Note that depending on the training corpus, the results are very different.

So far we have seen how to build a RNN and how to use them to build a language model. We'll continue covering how to build stronger models that can deal with some challenges such as vanishing gradients.

1.8. Vanishing gradients with RNN

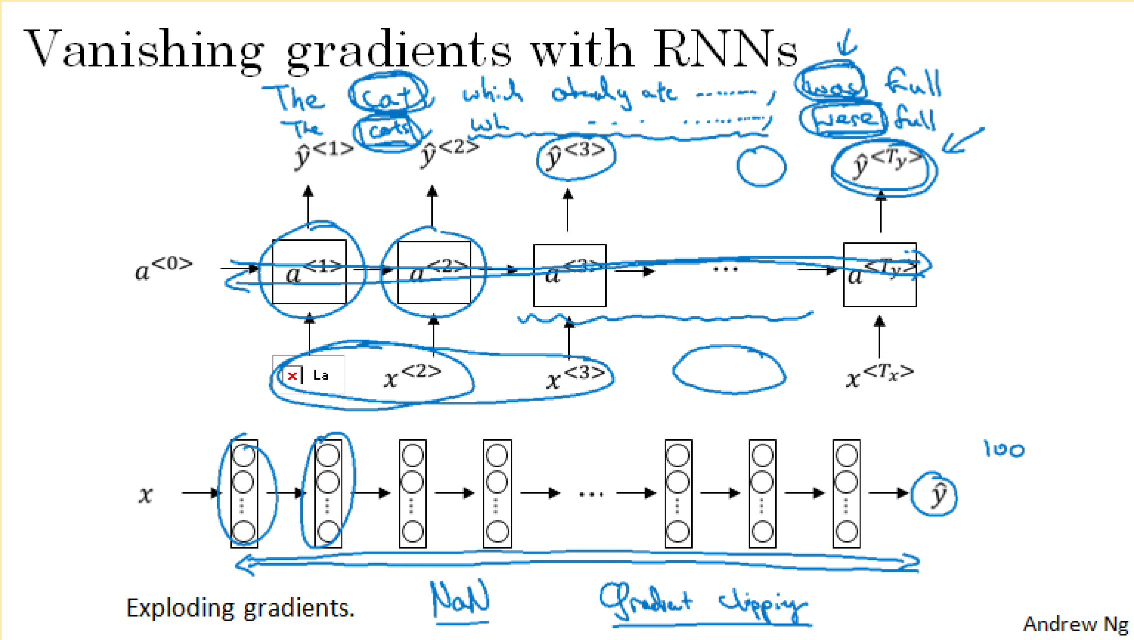
One of the common problems of a basic RNN algorithm is that it runs into vanishing gradient problems.

Let's have an example:

- *"The **cat**, which ate ……………, **was** full"*
- *"**The** cats, which ate ……………, **were** full"*

We see that there is a long term dependency between the bold words. But RNNs we've seen so far are not very good at capturing very long term dependencies.

When we discussed about very deep neural networks, we addressed the problem of vanishing gradients.



So, this is exactly the problem we have here (we have a deep neural network). It is really hard to backpropagate and maintain all the effects. In fact, the outputs are mainly influenced by values close to it.

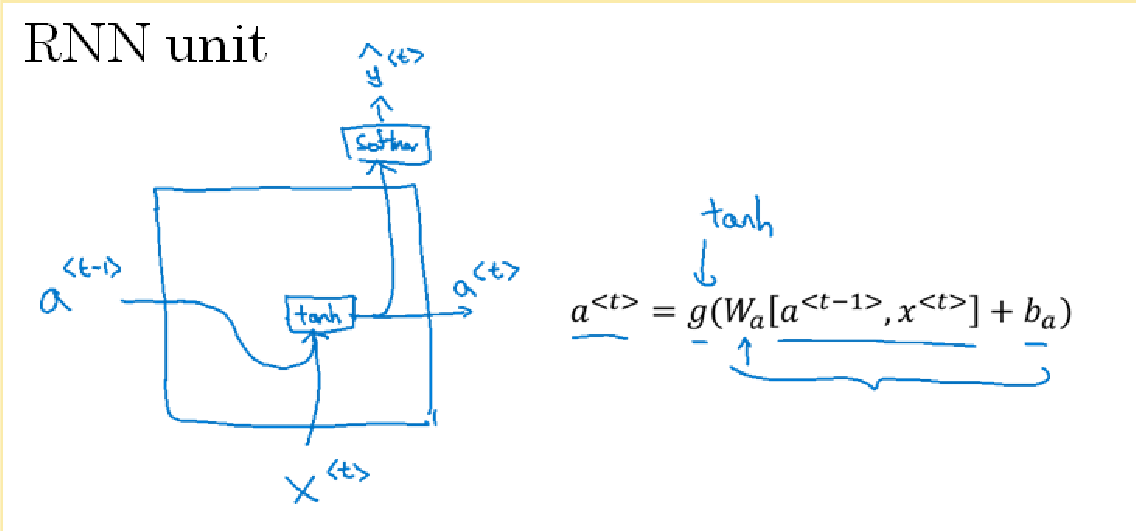
In addition, we can also have problems with exploding gradients, but vanishing gradients are more usual. However, to address exploding gradients we can use gradient clipping (setting an upper threshold so that the weight vector is rescaled if an upper threshold is exceeded). This is a robust solution.

So exploding gradients are easy to overcome, but it's not the same for vanishing gradients.

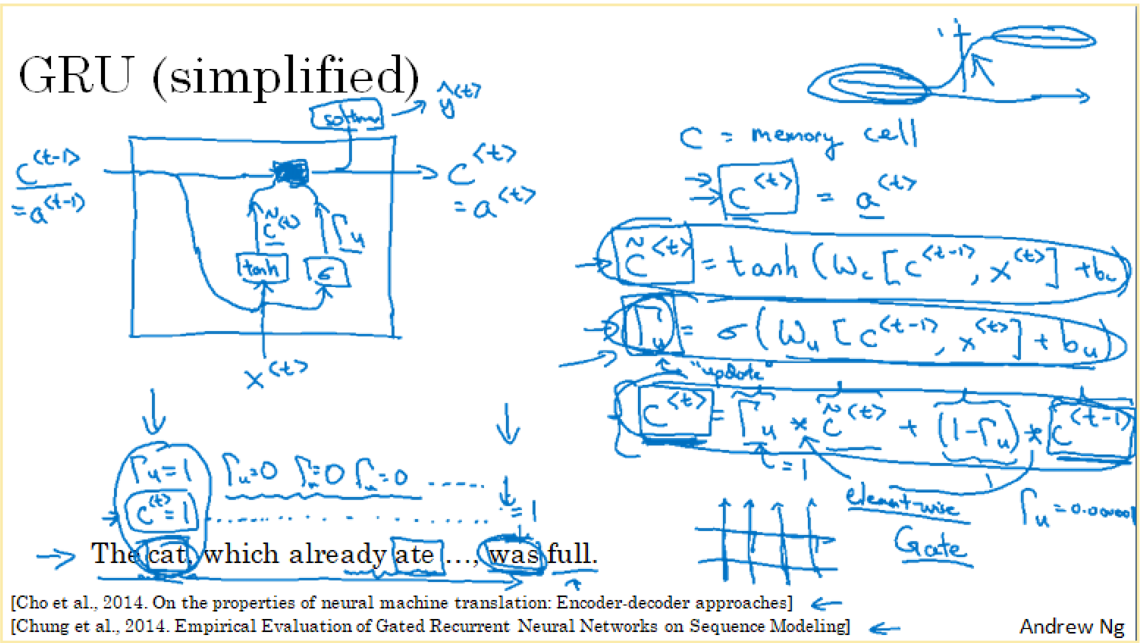
1.9. Gated Recurrent Unit (GRU)

The GRU 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.

The RNN unit as we have seen it can be explained with the slide below:



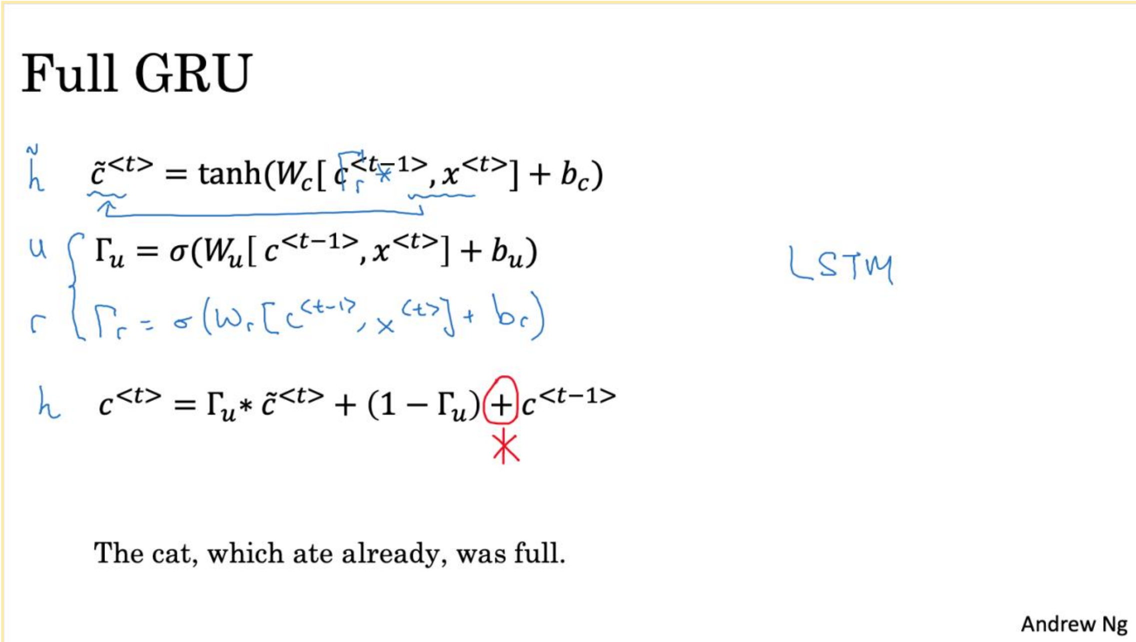
The GRU structure is as follows:



The intuition is the following: we introduce a new parameter Γ_u(gamma update) and the C (memory cell). To understand it, we can think of it as if the gamma was a "gate": it goes through the sigmoid activation function (so it takes values almost 0 or 1) and determines whether the memory cell C is updated with the incoming activation or not.

This way, we can make the patterns "travel" through the series and capture long term relationships without incurring in vanishing gradients (because if the weights are really near to zero, then C won't be updated).

This corresponds to a simplified GRU structure. The full GRU structure is as it is shown below:



We also introduce a Γ_r (we can think of r standing for relevance), and it tells us how relevant is $c^{<t-1>}$. We'll see more detail about this when we cover LSTMs.

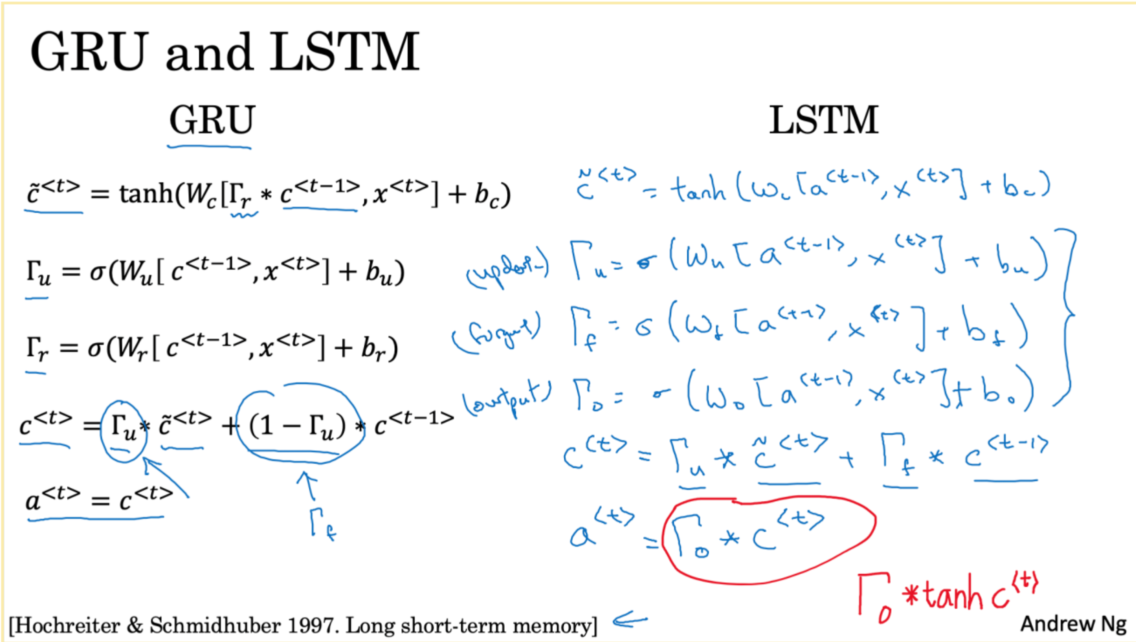
Throughout the years, researchers have proved that GRUs and LSTMs work really well in sequence problems. They differ in some ways, but they are based in the same ideas.

Let's cover the LSTM:

1.10. Long Short Term Memory (LSTM)

We have seen that the GRU allos us to learn very long range connections in a sentence. The other type of unit that allows us to do this very well is the LSTM (and it's even more powerful).

In the GRU, we have the two gates (update and relevance), and $\tilde{c}$ which is a candidate for replacing the memory cell, and then we use the update gate to decide whether or not to update c.



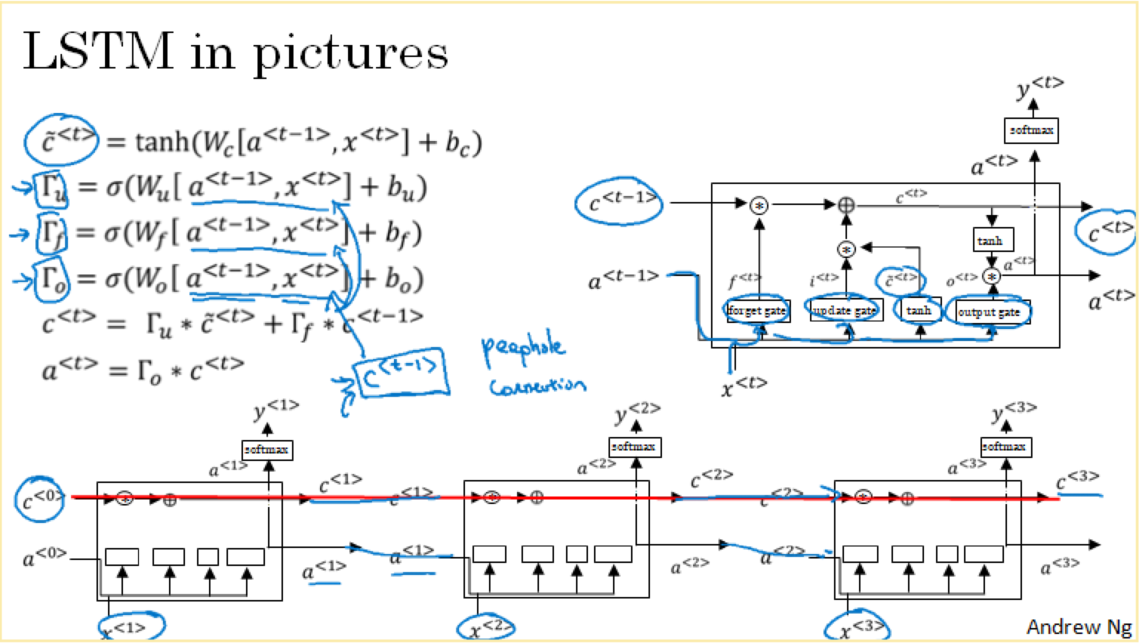
The LSTM is a more powerful and general version of the GRU. The first difference is that now $c^{<t>} \neq a^{<t>}$. The second thing is that we don't only have a single update term (so we won't use Γ_u and$(1 - \Gamma_u)$). Now we have a *forget* term. In addition, we have an *output* gate as well. So now, the equations are as follows:

LSTM units

GRU	LSTM
$\tilde{c}^{<t>} = \tanh(W_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c)$	$\tilde{c}^{<t>} = \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c)$
$\Gamma_u = \sigma(W_u[\, c^{<t-1>}, x^{<t>}] + b_u)$	$\Gamma_u = \sigma(W_u[\, a^{<t-1>}, x^{<t>}] + b_u)$
$\Gamma_r = \sigma(W_r[\, c^{<t-1>}, x^{<t>}] + b_r)$	$\Gamma_f = \sigma(W_f[\, a^{<t-1>}, x^{<t>}] + b_f)$
$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>}$	$\Gamma_o = \sigma(W_o[\, a^{<t-1>}, x^{<t>}] + b_o)$
$a^{<t>} = c^{<t>}$	$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>}$
	$a^{<t>} = \Gamma_o * c^{<t>}$

(Watch out with the mistake in the slide on the final equation.)

Let's see it in pictures:



Although it is a bit tricky, we can see that we get $a^{<t>}$ and $x^{<t>}$, pass them through the *forget*, *update* and *output* gates and use them to compute $c^{<t>}$ starting from $c^{<t-1>}$. In the red line we can see that it may be easy to a memory cell to travel through the network, so that's why LSTMs and GRUs are very good at memorizing certain values even for a long time.

This LSTM may have some variations. Some people include the *peephole connection*, which consists in adding $c^{<t-1>}$ as seen in the slide.

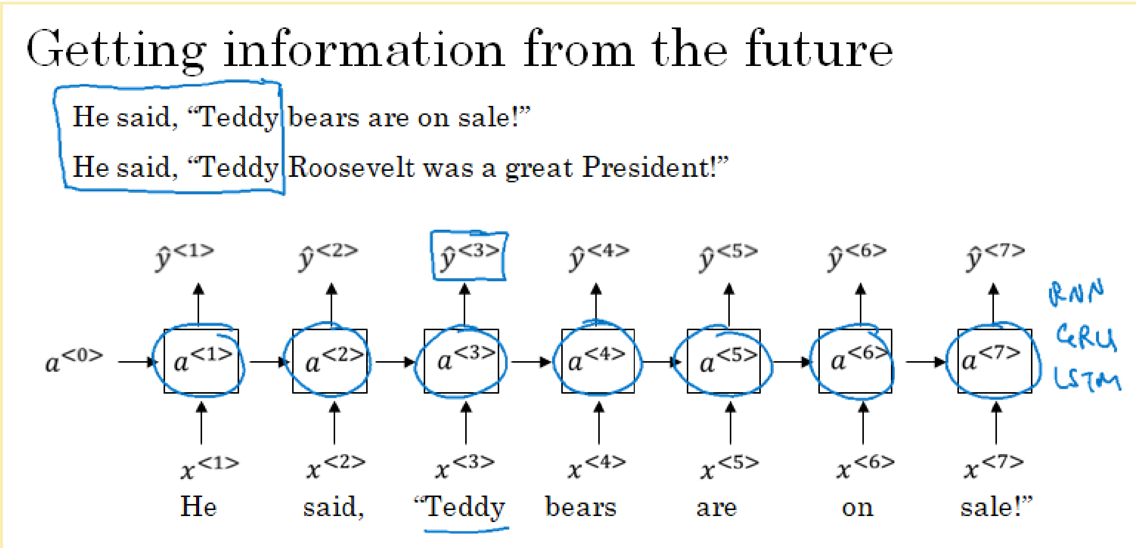
So, when should we use LSTMs and when GRUs? There is no consensus with this: on some problems, LSTMs will perform better and on others, the GRUs will do.

Needs to be said that the GRUs are simpler and therefore it's easier to build a bigger network. However, with either GRUs or LSTMs, we'll be able to build neural networks that can capture much longer range dependencies.

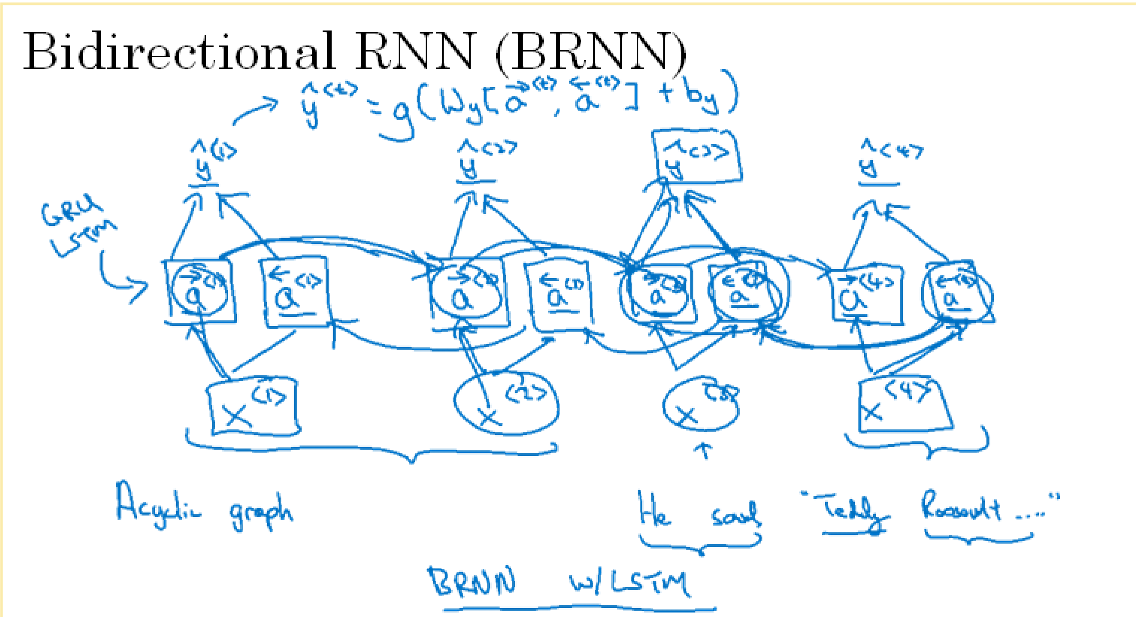
By now, we have seen most of the cheap building blocks of RNNs. But there are two more ideas that will let us build much more powerful models: bidirectional RNNs and Deep RNNs.

1.11. Bidirectional RNN

Sometimes, we need to be able to look into the future to capture the real meaning or characteristics of a sentence. See an example in the next slide:



These blocks only work in the forward direction. What bidirectional RNNs do is:



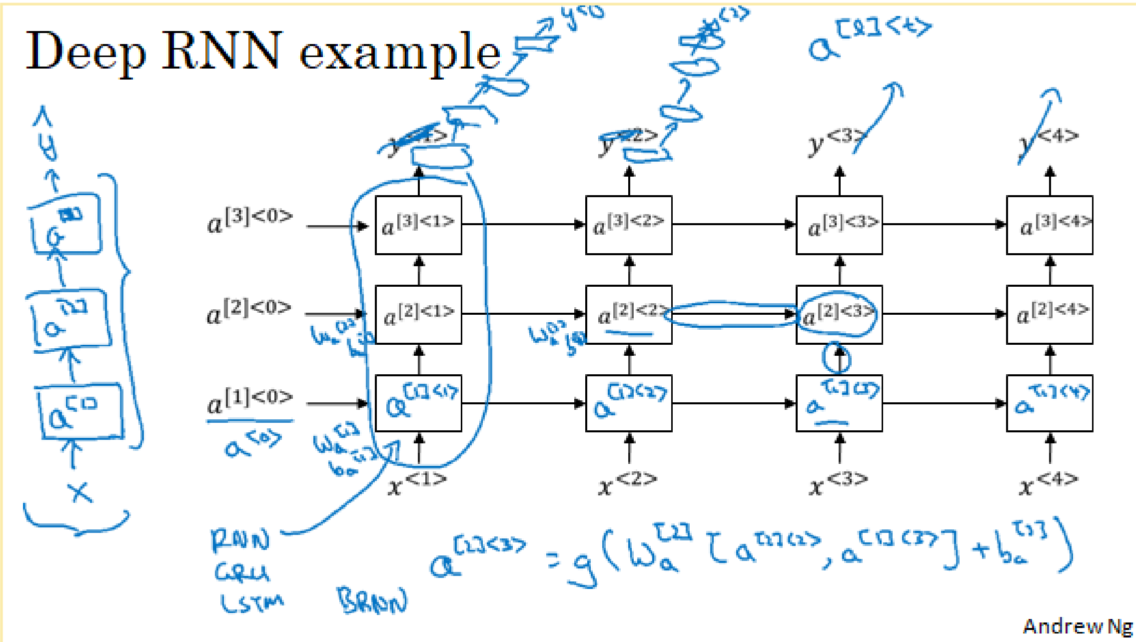
The idea is that now, for each cell, we have two layers (forward and backward sense) and we make the predictions in both senses in the sentence.

For example, if we want to know if Teddy is a part of a person's name, we can now look before (*he* said) and after (*Roosevelt*) in the sentence.

The disadvantage of bidirectional RNNs is that you need the whole sentence to make the predictions. For example, if we are building a speech recognition system, this might be a problem since we need to wait for the person to stop talking.

1.12. Deep RNN

The different RNN versions we have seen will already work quite well by themselves. But if we want to increase performance, we can stack multiple layers of RNNs like in the following slide:



Note that intuitively, the layers are now horizontal and the sense of going through time is vertical. Note that we don't usually see too many deep networks since they are expensive to train.

2. Natural Language Processing & Word Embeddings

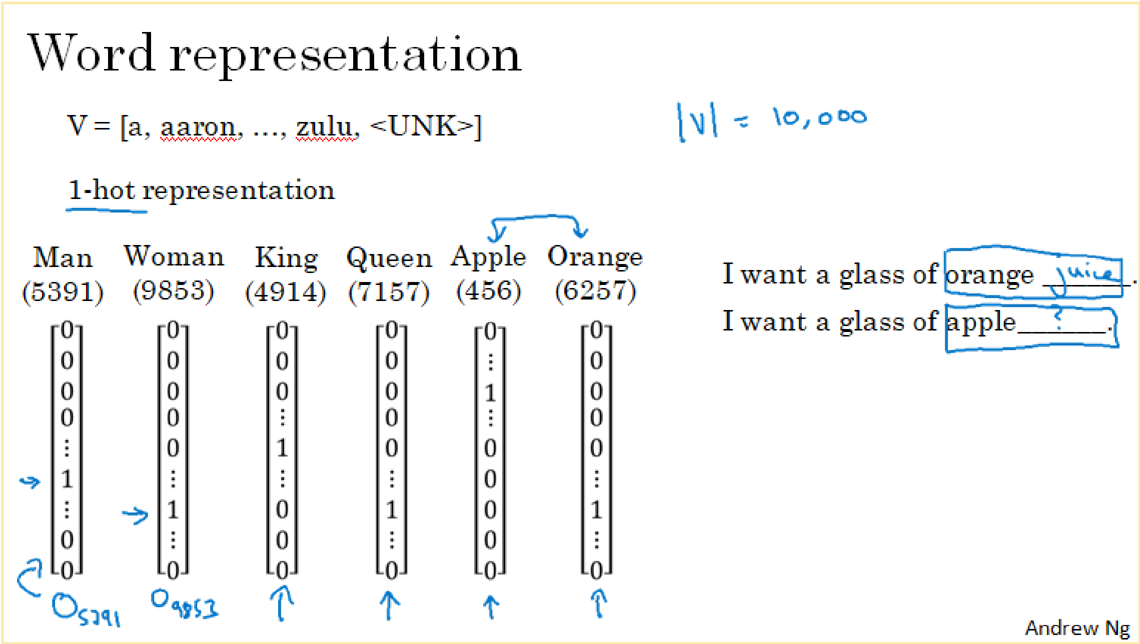
We have learned about RNNs, GRUs and LSTMs. Now, we are going to see how many of these ideas can be applied to NLP, which is one of the features of AI because it has been revolutionized by deep learning. One of the main ideas is the use of words embeddings.

2.1. Introduction to Word Embeddings

First of all, we'll see how to represent words:

2.1.1. Word Representation

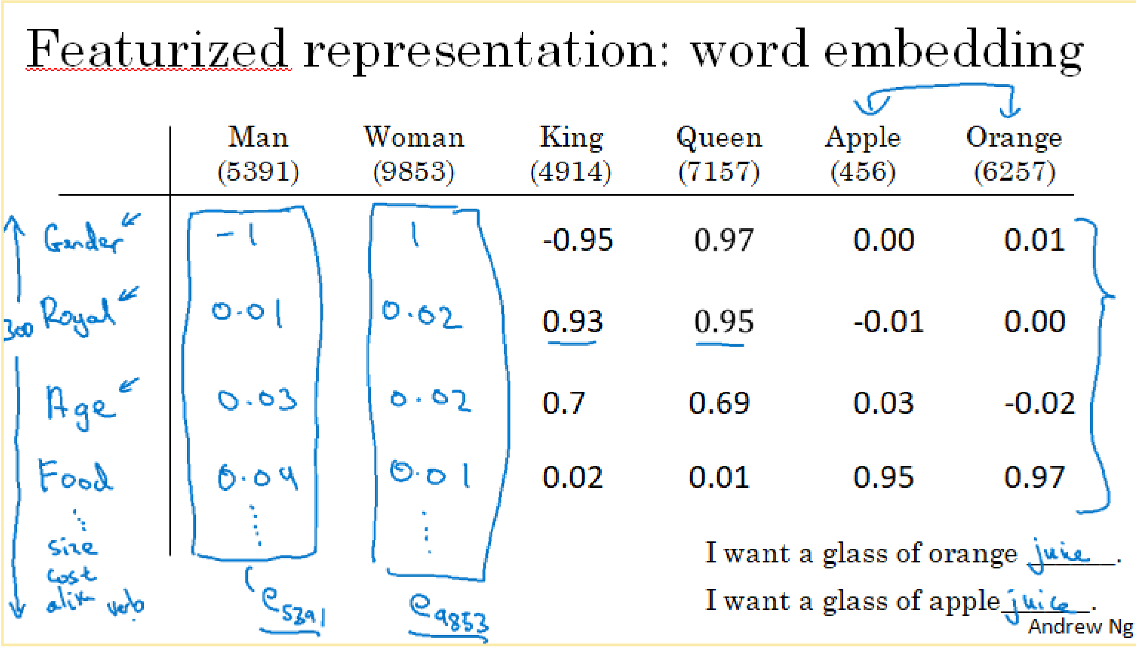
So far, we have represented words using a vocabulary of words (let's say 10,000 words) and a one-hot vector:



One weakness of this representation is that it treats each word as a thing in itself, and it doesn't allow an algorithm to easily generalize the cross words. For example, if we have the sentence "*I want a glass of orange* ___", it is likely that it ends with *juice*. However, if we have the same but with *apple*, since there is no relationship in the representation between *orange* and *apple*, the algorithm won't learn anything from that.

So, every word is at the "same" distance from the other ones. In addition, the vector product between two words is always zero.

An improvement of this is a **featurized representation**: each word can be represented by different features:

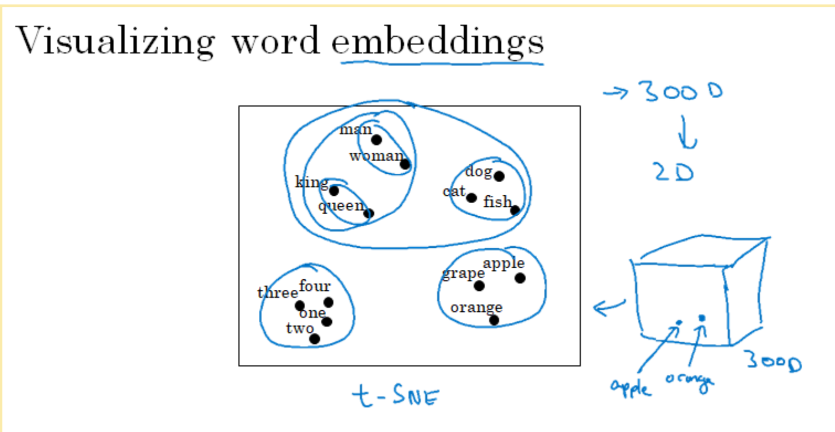


Note that now, instead of denoting each word with o_{5391} (for one-hot), we'll denote them with e_{5391}.

We can see that, if we compare the vectors of *apple* and *orange*, now they will be more similar. They will differ in things such as the color, but will also have a lot of things in common.

We'll see how to learn these word embeddings. But we'll see that the features we'll be learning are not as intuitive as we have seen and will be harder to figure out. However, it will allow the algorithms to quickly figure out relationships between words.

In addition, one common thing to do is, once we have learned the feature representation, we can plot it in 2D or 3D:



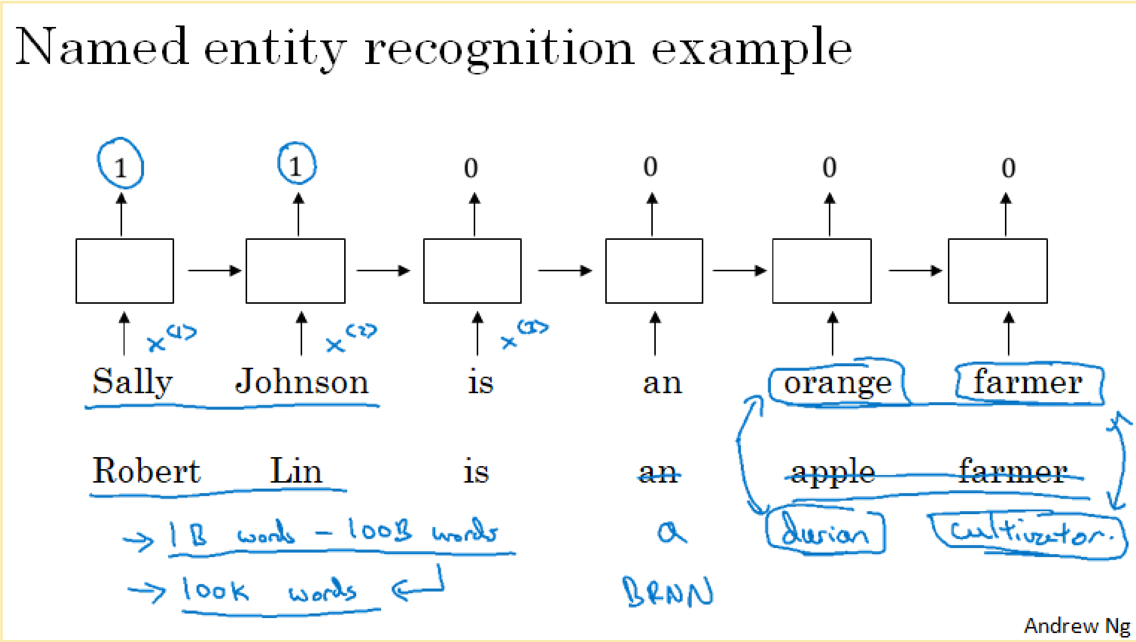
With *t-SNE*, we can see that similar words will be together.

So these representation of words are called **embeddings**. This is because we create a space of *X* dimensions and then embed each word to a point of it.

So, we have seen why we would prefer to learn word embeddings. Now, let's see how to use them in NLP algorithms.

2.1.2. Using Word Embeddings

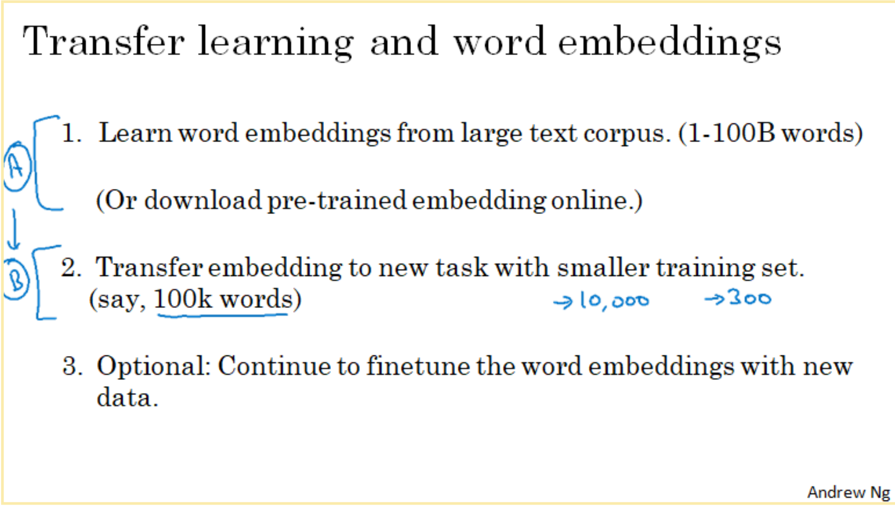
Let's continue with a Named Entity Recognition example:



Let's say we want to detect the names of persons. If we have the sentence "*Sally Johnson is an orange farmer*", the way of knowing that Sally Johnson is a person is because we know that an orange farmer is a person. So now, if we input the sentence "*Robert Lin is an apple farmer*", if we have well-trained embeddings, we'll know that *apple farmer* is also a person so it will be easy to detect the named entity.

One really interesting thing is that, if we have a rare word such as "*a durian cultivator",* which is not in our training corpus, we still can get the relationship if the embeddings are trained well.

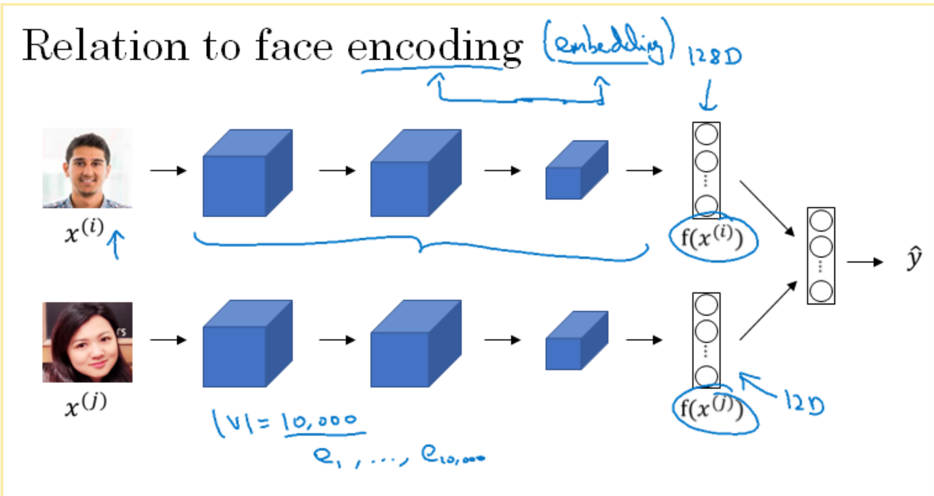
And this is how it works: we can use transfer learning and download from the internet embeddings that are composed of 1-100 billion words, and then use them to represent the words in our example and train the network to get the predictions in our corpus' example (let's say we have 100k words):



In addition, we can also try to fine tune the word embeddings with our new task's data. But we should do this only if we have a really big dataset.

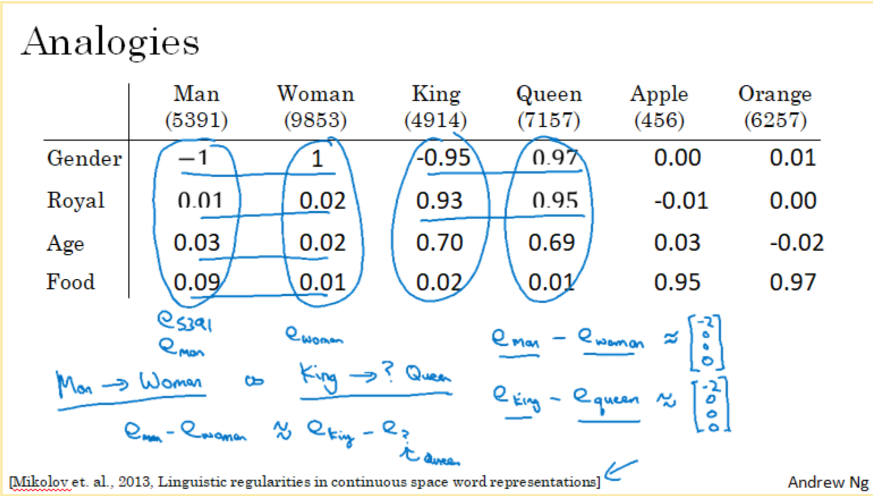
So, word embeddings make the difference when we have a relatively small training set.

Note that this has a relationship with what we saw in course 4 related to face encoding.

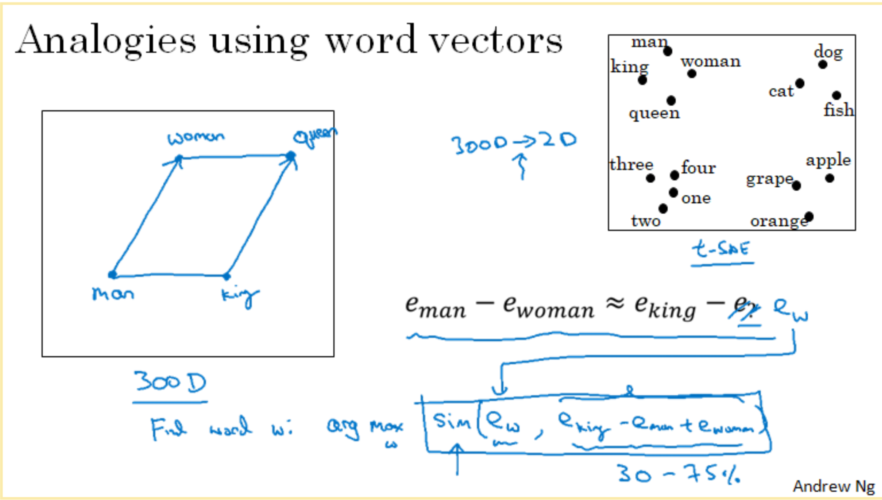


2.1.3. Properties of Word Embeddings

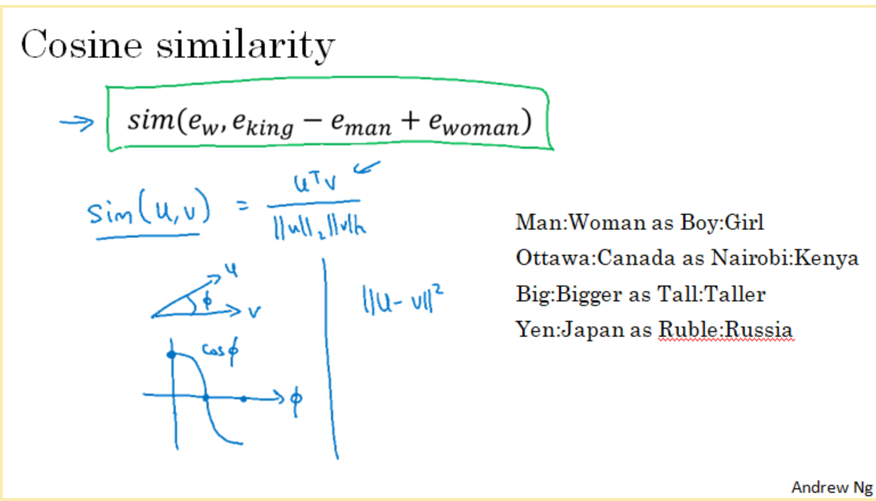
One interesting property of the word embeddings is the analogy reasoning:



Given three words, we can get the 4th word that suits into the comparison:



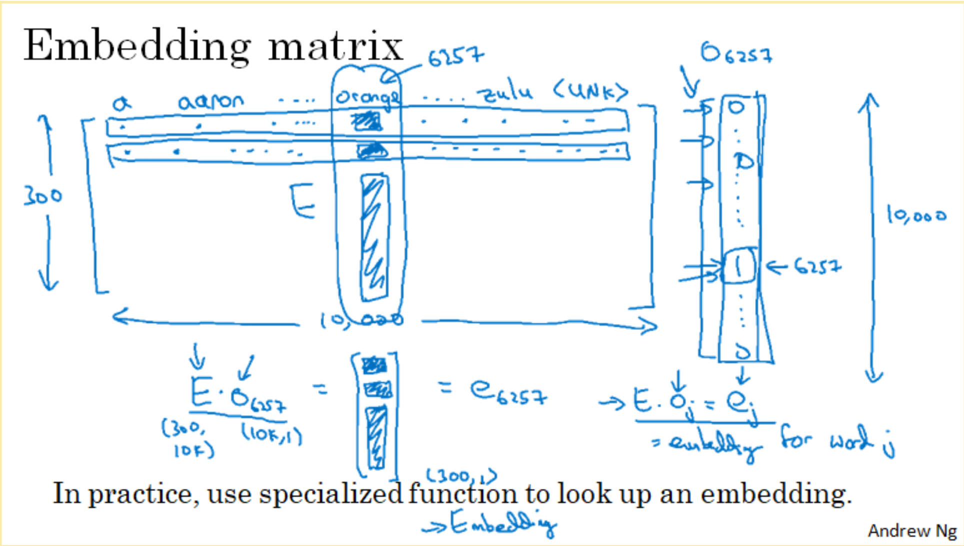
The most commonly used similarity function is the cosine similarity because it works quite well for this analogy reasoning.



Let’s see how to actually learn these word embeddings:

2.1.4. *Embedding matrix*

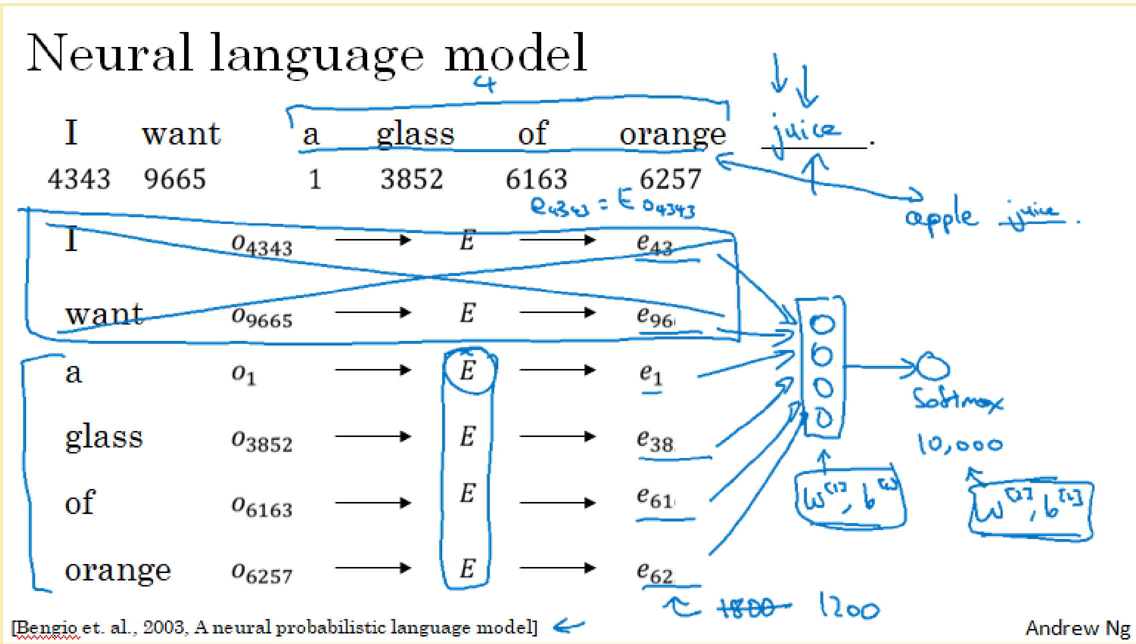
When we implement an embedding learning algorithm, what we end up learning is an **embedding matrix** E.



Note we can obtain $e_j = E \cdot o_j$.

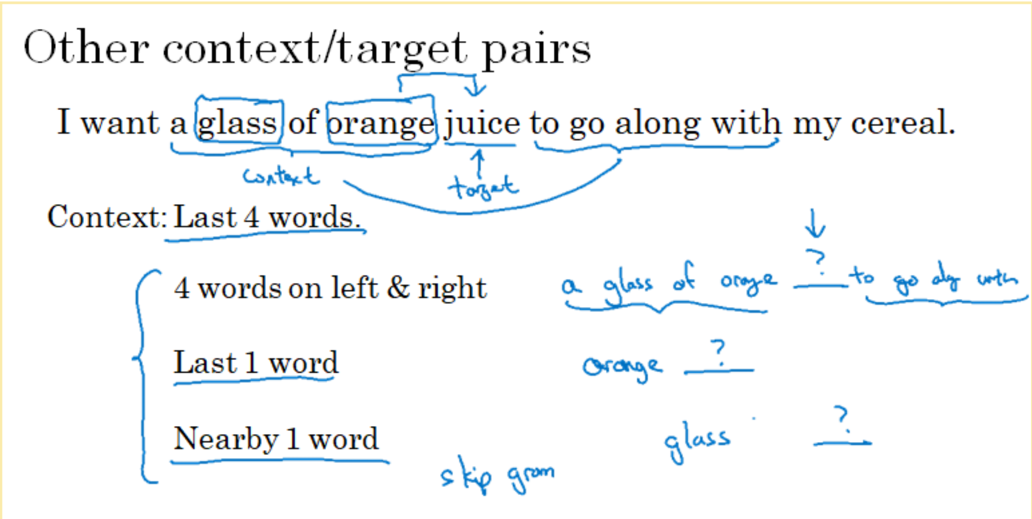
2.2. Learning Word Embeddings

We’ll start with some complex algorithms that will help us understand how to learn word embeddings and then show the simplified versions that work pretty well.



The idea of the neural language model is to use windows of n words and then try to predict the following one (and to learn, we compare to the complete sentence). If we do this through all the examples and all the different windows (we keep moving them, although n is an hyperparameter), we will be able to learn the embedding matrix E.

Let's see how to derive an even simpler algorithm:

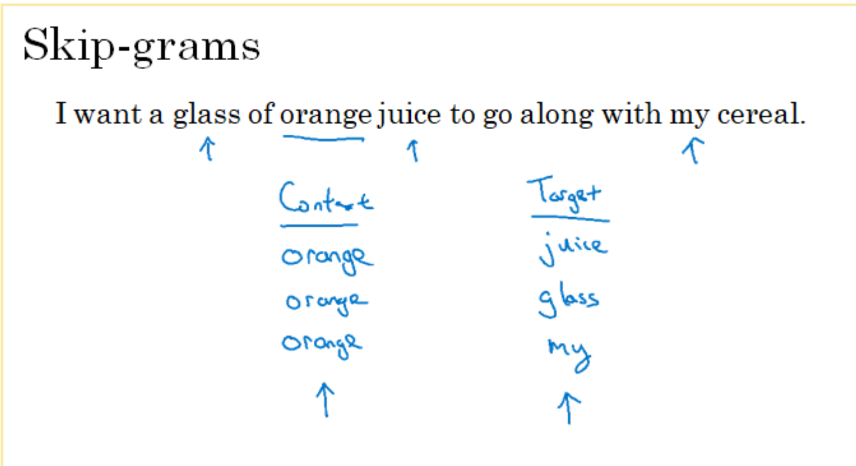


We can use the context around the word we're trying to predict to learn the embeddings. This context can be as simple as one word (the nearby one). This is the idea of Word2Vec:

2.2.1. *Word2Vec*

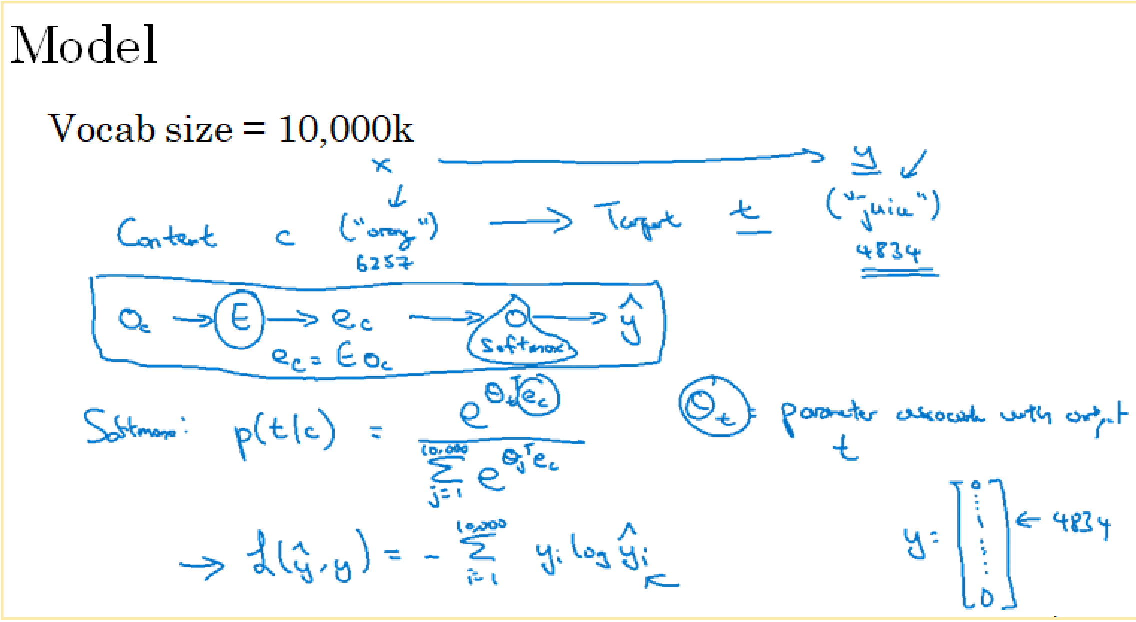
The Word2Vec algorithm is a simple and more efficient way to learn word embeddings. It is a skip-gram model.

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:



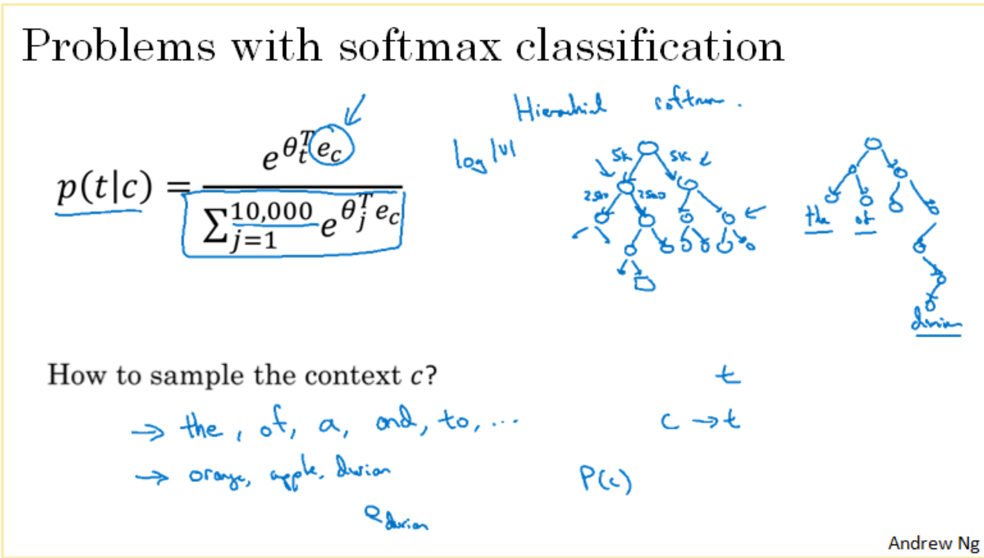
This is, instead of having a fixed context window, we'll randomly pick words to be the context words from a window (+-5 or +-10 words).

So, here are the details of the model:



So, we start from some context c and we want to map it to the target t with a softmax unit. In the process we'll learn the E matrix along with the *theta* parameters of the softmax.

It turns out that there are a couple problems with this algorithm. The primary one is computational speed, since we have to go through the whole vocabulary to compute every probability. One solution to this is to use a hierarchical softmax classifier, which carries on splitting the vocabulary.

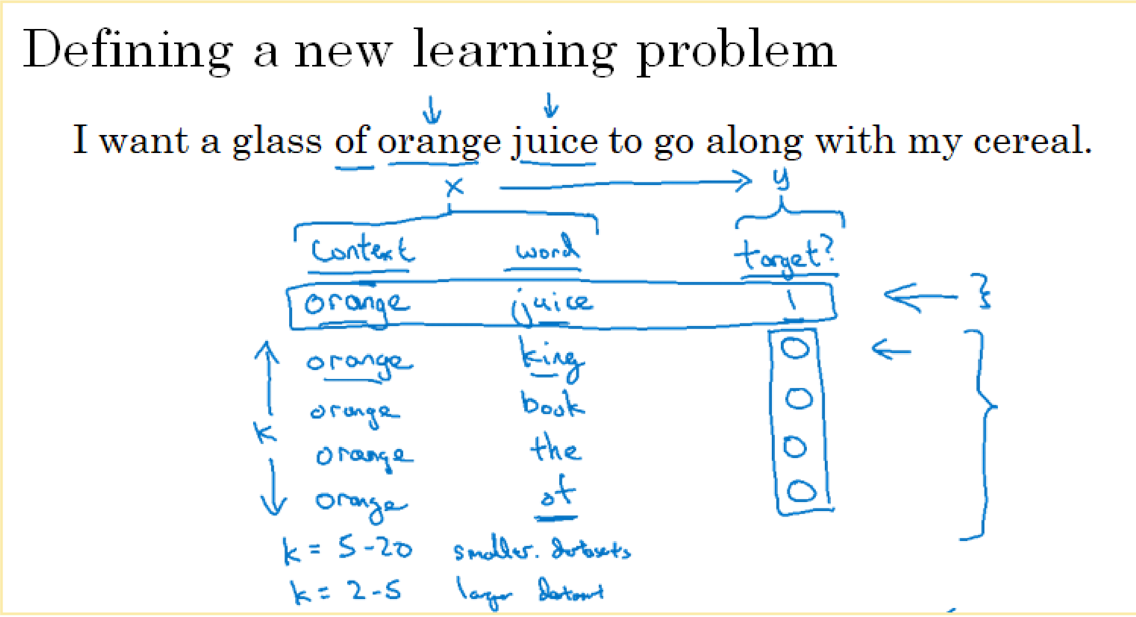


Another thing we need to discuss is how to sample the context c. In practice, the context is not taken randomly, but we try to balance and reduce the "weight" of very common words such as articles or prepositions.

As we have seen, the biggest drawback of the skip-gram model is the computational cost. Let's see how to reduce it with negative sampling.

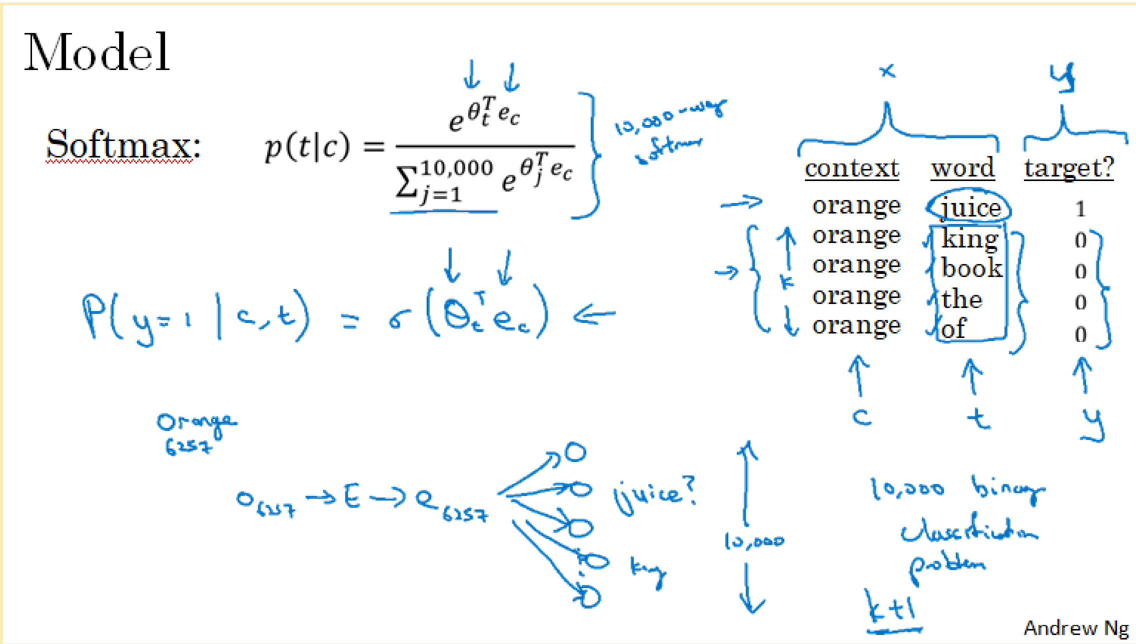
2.2.2. Negative sampling

We're going to redefine the problem. So, we are going to predict for each context-word pair, if it is a context-target pair:



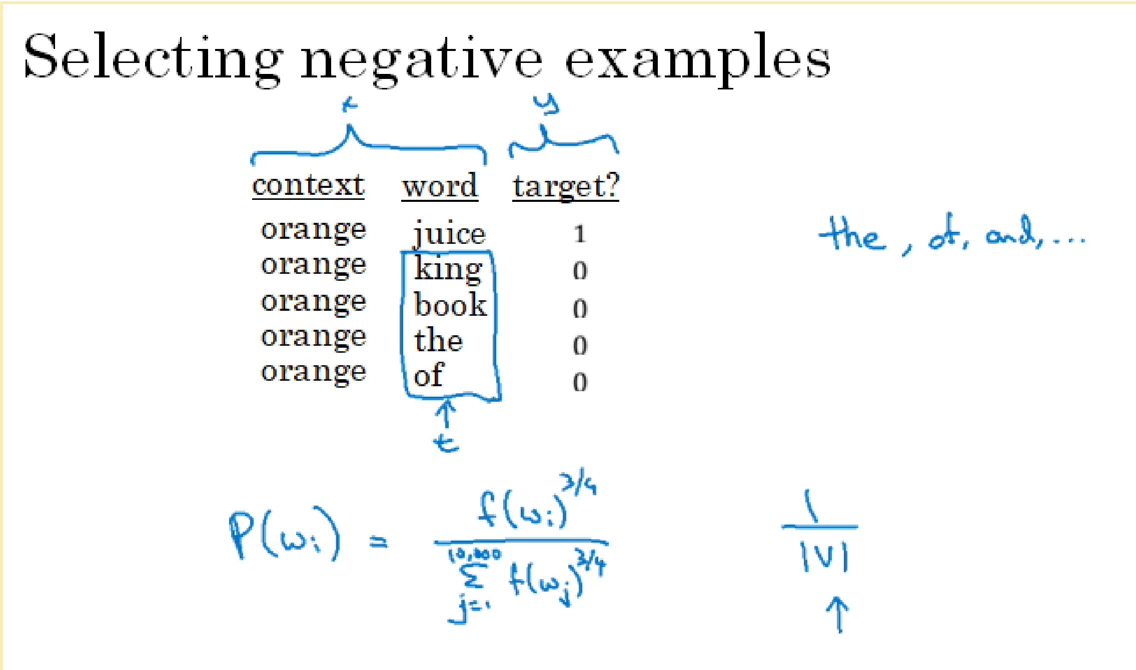
We'll construct the problem by getting a *context* word, randomly putting it together with different words for *k* times and then labeling the relationship. The *k* parameter can change depending on the size of the dataset.

Then, once we have *X* and *y*, let's define the model to capture the relationship:



So, the network will have 10,000 units (same as vocabulary length), but on each iteration we will only train *k+1* of them (*k* stands for the number of negative examples and 1 is the positive example).

One last thing before wrapping up is how to choose the negative examples:

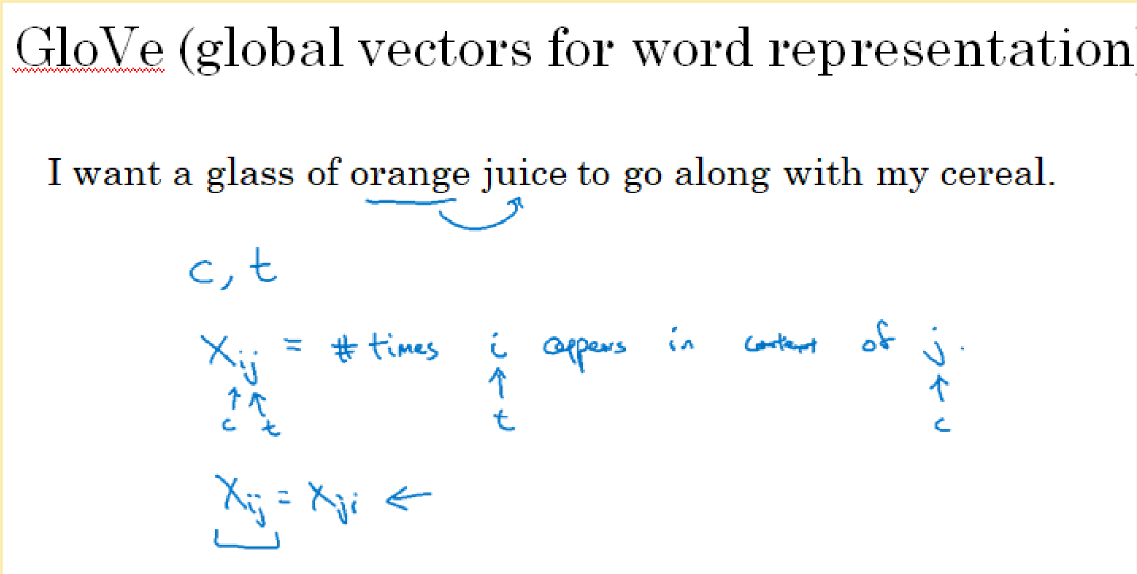


The above heuristic seems to work decently well.

Now, we are going to see a different embedding learning algorithm that is maybe even simpler that what we've seen so far.

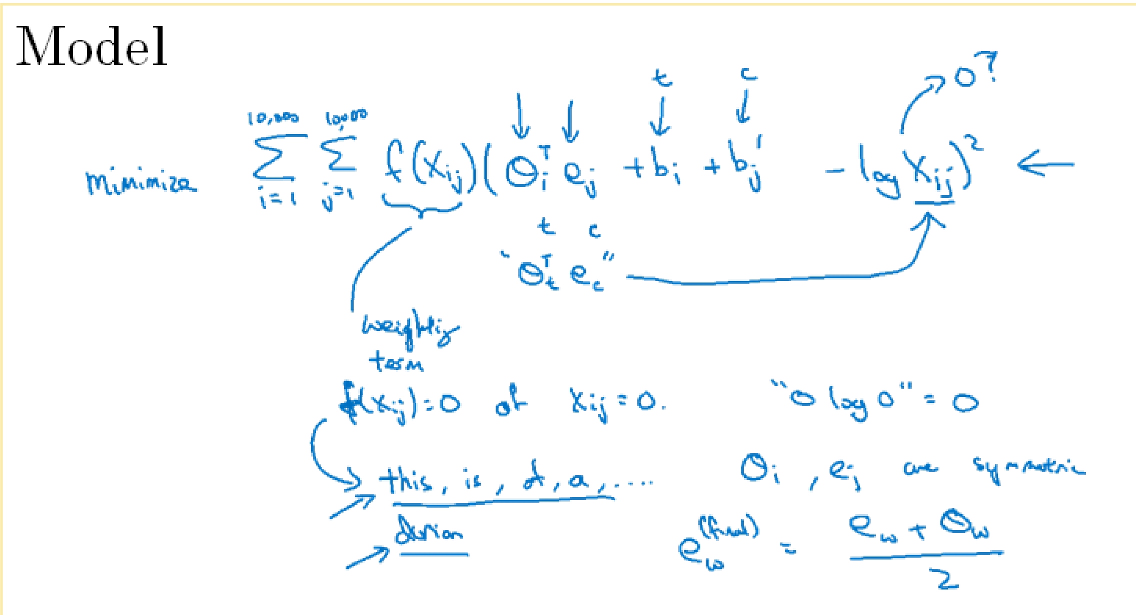
2.2.3. GloVe word vectors

Previously we were sampling pairs of words (context and target words) by picking two words that appeared in close proximity to each other in our text corpus. The idea is maintained:



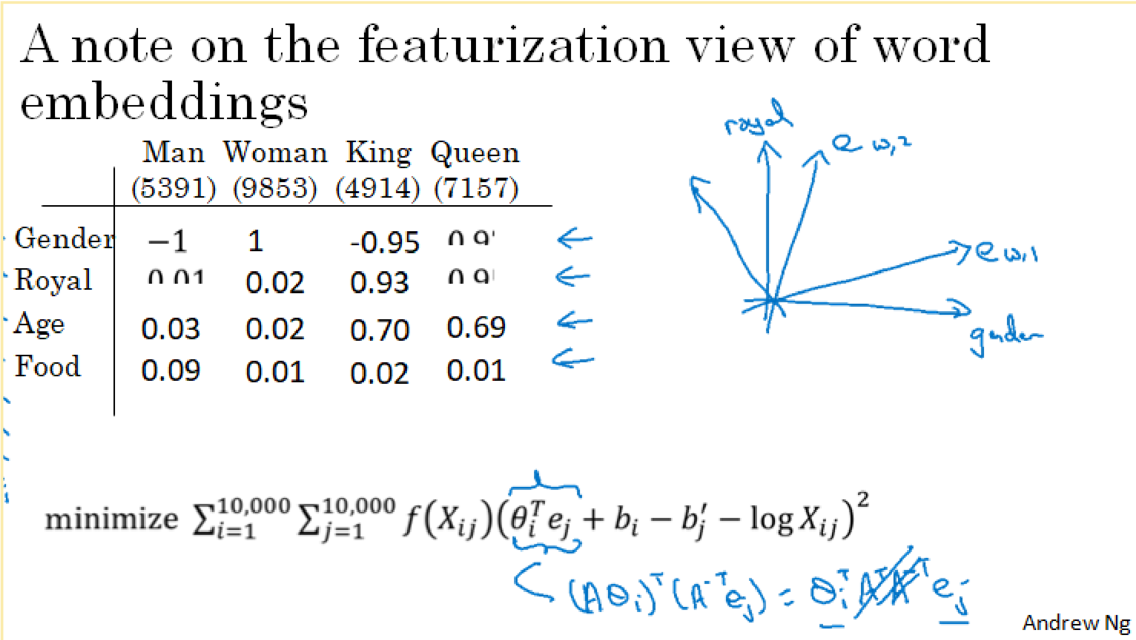
X_{ij} is the number of times i appears in the context of j (this is, a count that captures how often do words i and j appear close to each other.

So the model is:



Intuitively, we are going to minimize the difference between the real X_{ij} and the embeddings (network coefficients). This turns out to be a really simple yet powerful model.

To end with embeddings, there is a property that we should discuss. When learning word embeddings with the procedures we have seen, we cannot guarantee that the individual components of the embeddings (such as gender, royal, etc… in the slide) are interpretable.

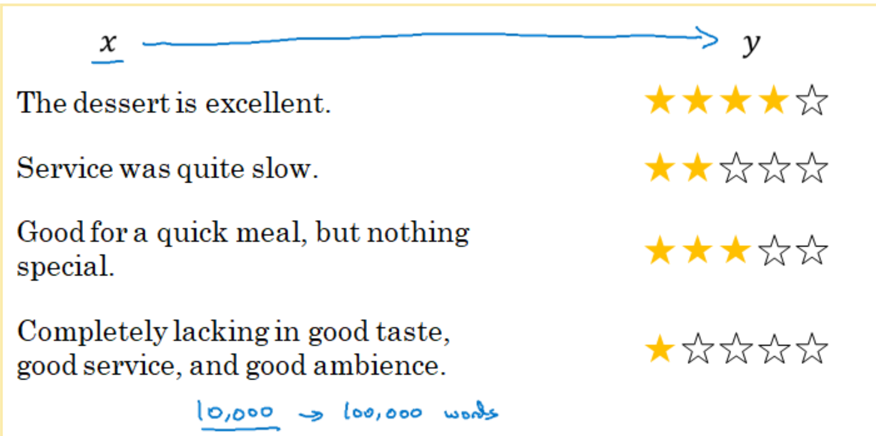


2.3. Applications using Word Embeddings

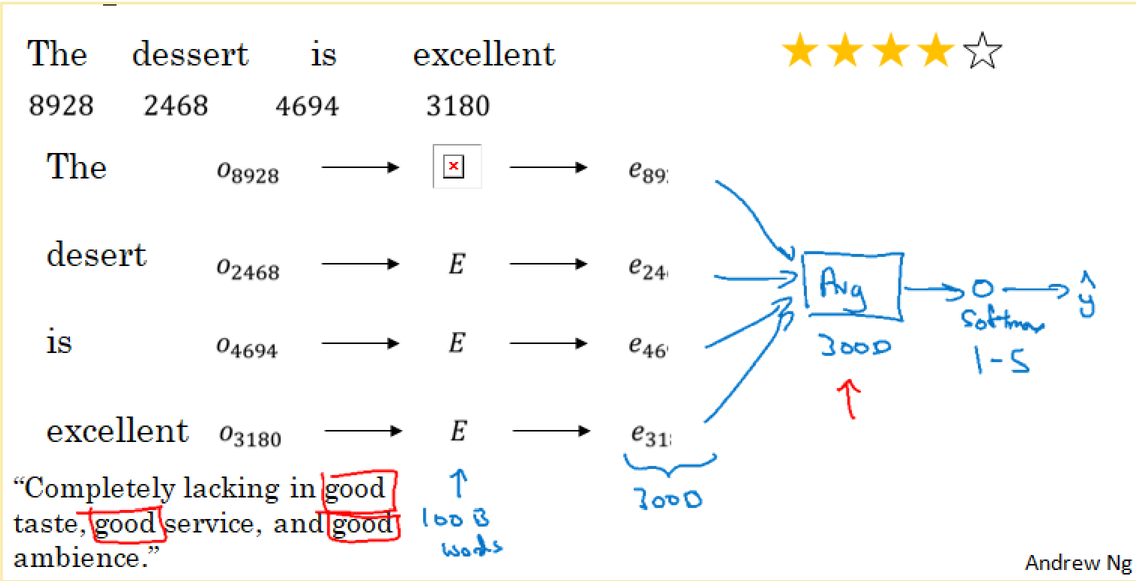
We'll see an example of **sentiment classification** using word embeddings. One of the common challenges of sentiment classification is that we may not have a really large dataset, but with word embeddings we can build good sentiment classifiers.

2.3.1. Sentiment classification

Here is an example of sentiment classification:

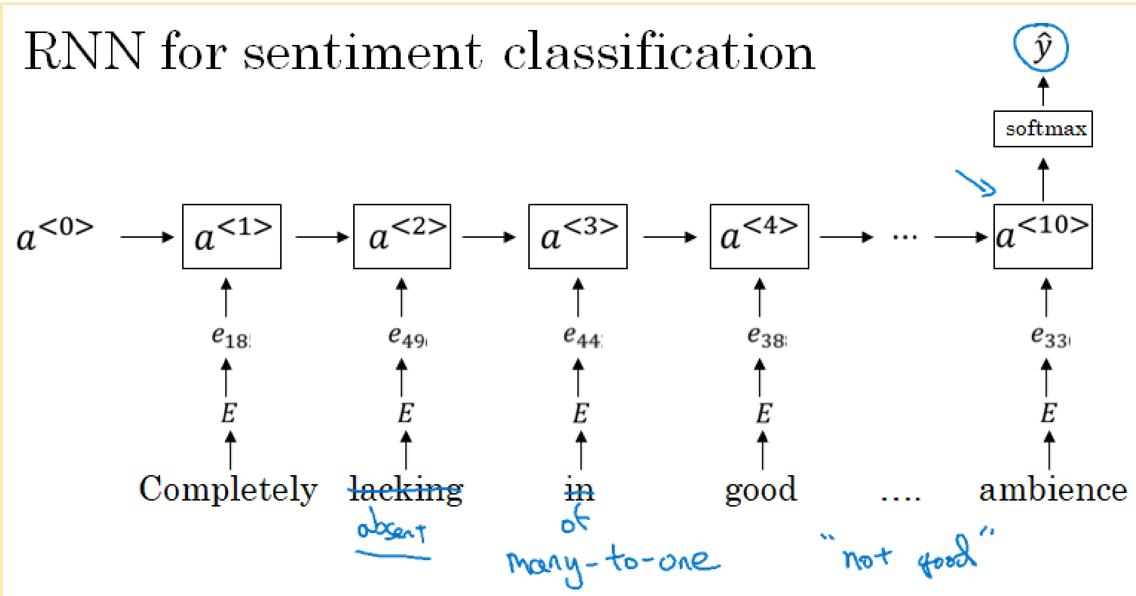


A simple sentiment classification model could be:



We can take the embedding vectors *e* and just sum or average them, which will give us a new feature vector. Then, we can feed it to a softmax unit to get the prediction.

The problem is that we are ignoring word order. So for example, a bad review which has the *good* word several times in it (see slide) could be misclassified. To solve this, we can use a RNN:



So, the conclusion is that we can use pre-trained word embeddings to get a better representation of our text (instead of one-hot vectors or something like that) and then, once we have the features for our text, feed them into a model and train it ourselves. This allows us to include much more "information" in our model.

2.3.2. Debiasing word embeddings

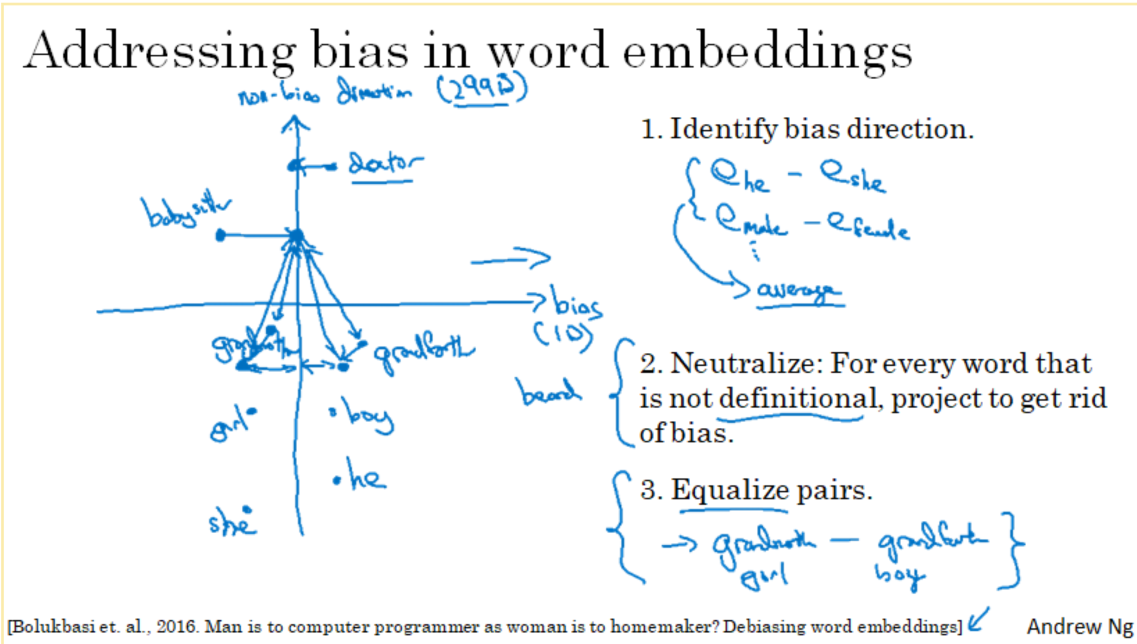
Word embeddings can reflect the gender, ethnicity, age, sexual orientation and other biases of the text used to train the model. For example, the following examples have been found:

- Man is to computer programmer as woman is to homemaker.
- Father is to doctor as woman is to nurse.

So we would like the embeddings to be neutral against these kind of things and return:

- Man is to computer programmer as woman is to computer programmer.
- Father is to doctor as woman is to doctor.

To address this, we can follow simple ideas as it can be seen in the below slide:

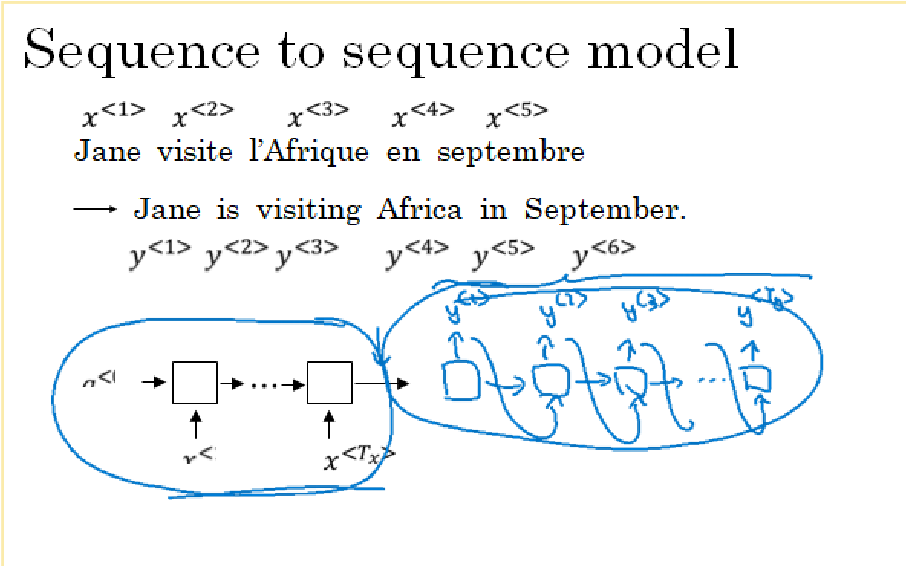


3. Sequence models & Attention mechanism

At this point, we are going to cover sequence-to-sequence models and then wrap up dealing with speech recognition and audio data.

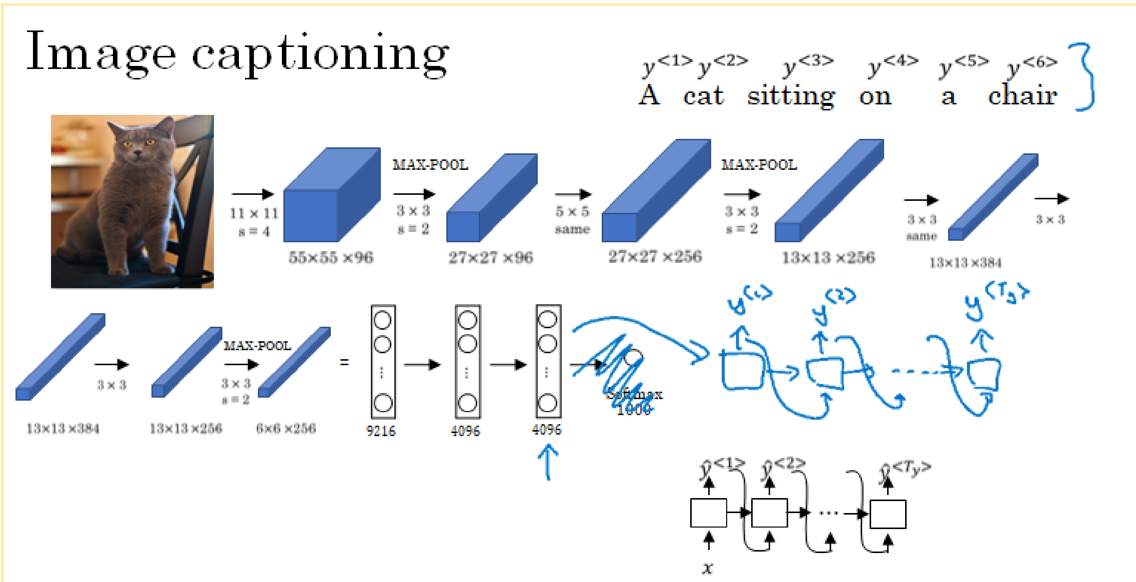
3.1. Various sequence-to-sequence architectures

Let's start with an example: a text translator:



By building a model like in the slide above (a first sequence model acting as an encoder and then a second model acting as a decoder), it turns out that it works pretty well.

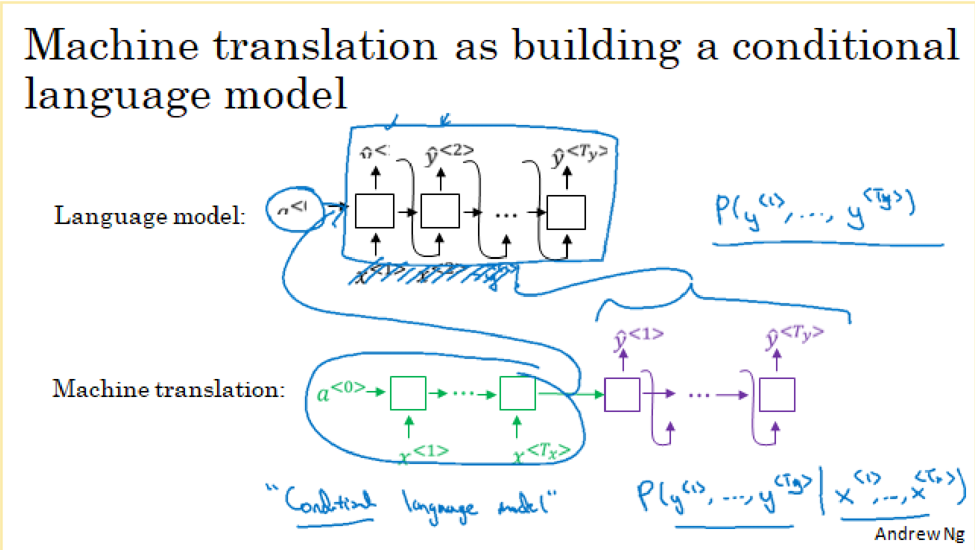
An architecture very similar to this also works for image captioning:



This is the simple idea of a sequence-to-sequence model. Let's see how to get them to return the most likely translation or caption.

3.1.1. Picking the most likely sentence

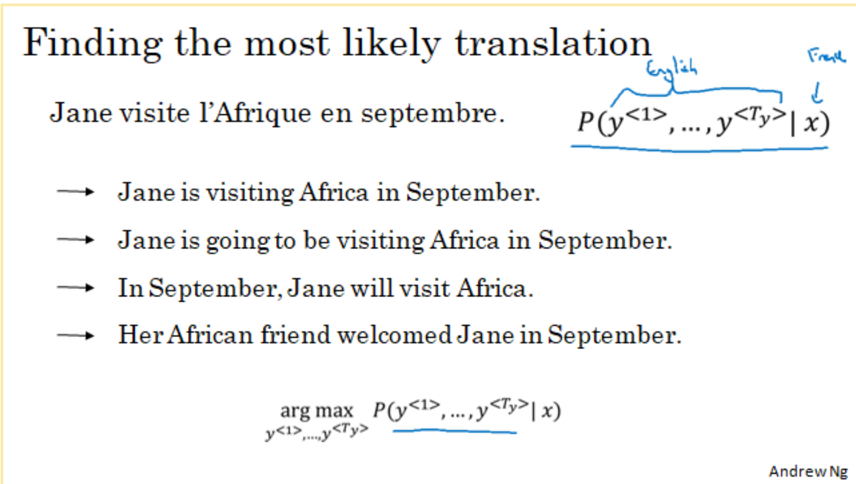
Let's start with an example: a text translator. The machine translation problem can be seen as building a conditional language model. As we saw previously, a language model outputs the probability of a sentence:



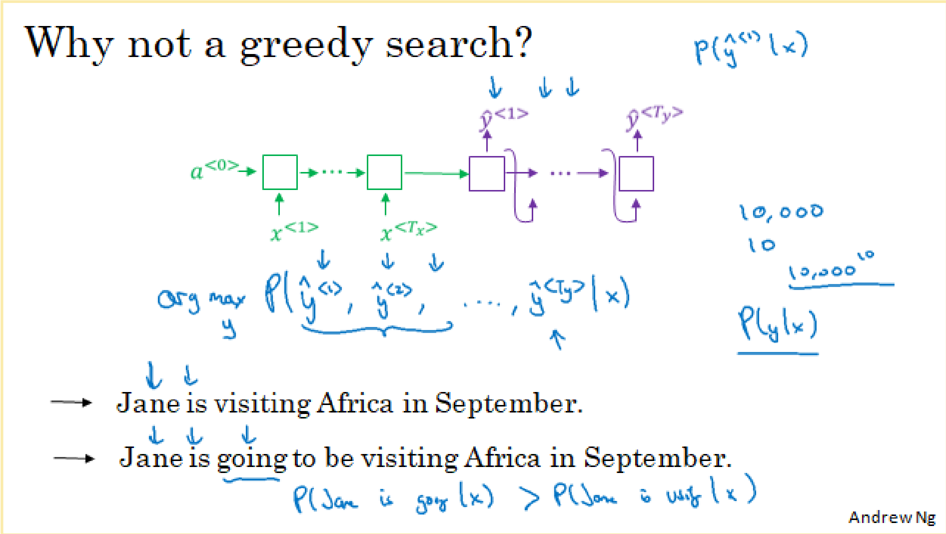
In machine translation, we have the same architecture as in the language model, but the input is an encoded network that figures out some representation for the input sentence.

So now, instead of outputting the probability of a given sentence, we can say, as an example, we'll output "the probability of an English translation conditioned on an input French sentence".

So, with an input French sentence, the model will tell us the probability of different translations:



And we'll try to get the translation with the maximum probability. To do this, we¡ll try to avoid a greedy search since we want the sentence in which the global probability is the biggest:



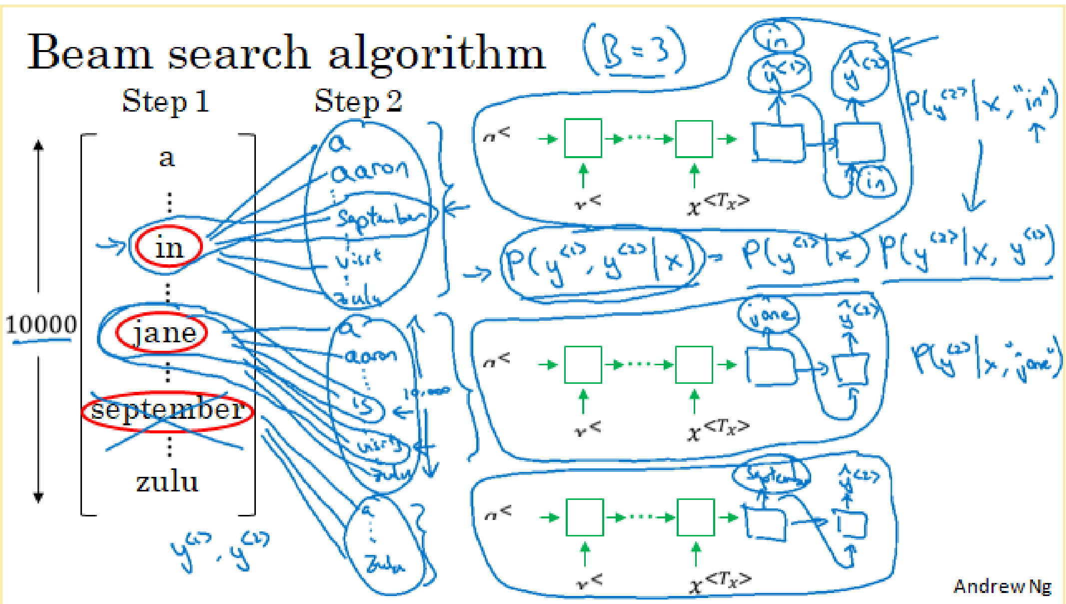
This means we have to find a search algorithm that allows us to search over all the possible translations and output the best. We'll do this with a beam search algorithm.

3.1.2. Beam Search

The greedy search algorithm considered the best option at each step. The beam search algorithm can analyze different alternatives depending on the beam width (3 in the example).

So, for example, if we have 10,000 words in the vocabulary, we'll run the net (encoding and decoding parts) and retain the 3 best words (the ones that have the higher probability).

With each of the three words, we'll run the net taking each word as an additional input apart from the encoding and output the pair of words that has the maximum probability. And so on.

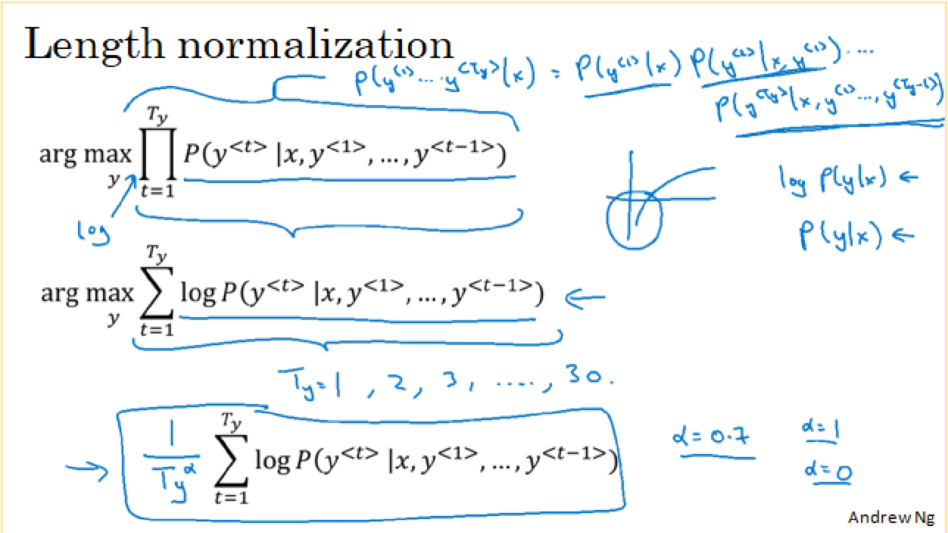


Finally, we should recall that a beam search with a beam width equal to one is a greedy search.

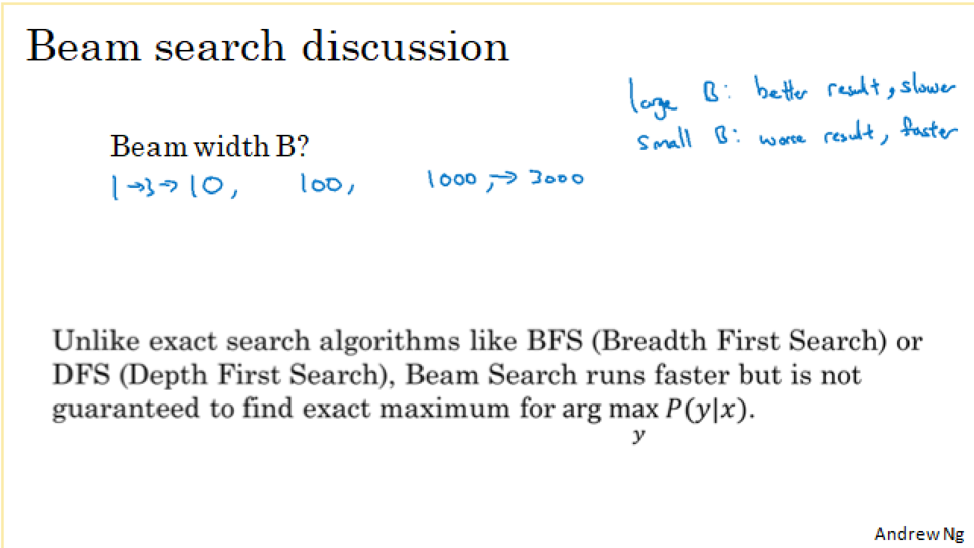
There are some tricks to make Beam Search work even better.

3.1.3. Refinements to Beam Search

To avoid numerical underflow, we will take logarithms and apply **length normalization**:



Regarding the beam width:

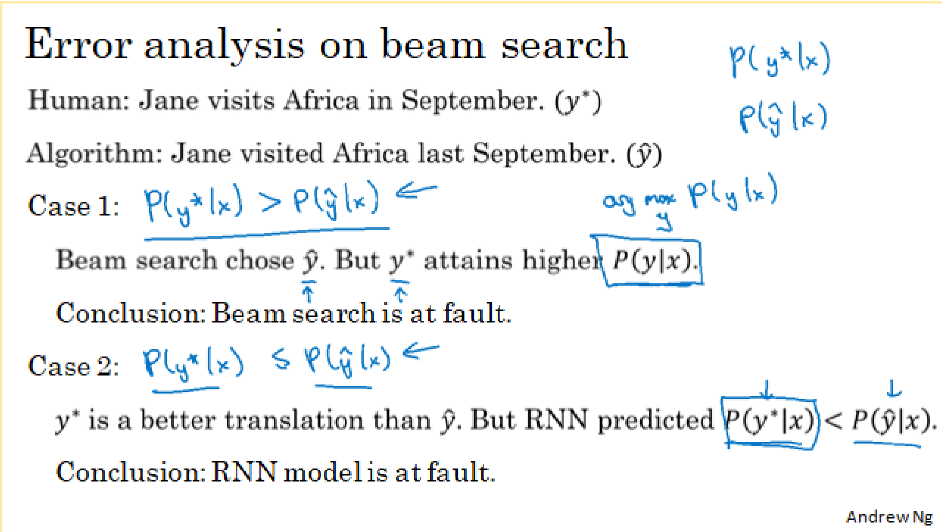


It turns out that beam search is a very good tool for error analysis.

3.1.4. Error analysis in Beam Search

Beam Search is a heuristic algorithm which can output a sentence that in fact is not the best one. So how can we figure out whether our RNN model is causing problems or it is the beam search itself?

Basically, we'll get a human translation and get its probability, and compare it to the probability of the sentence output by the model:



If we do this for a number of sentences, we can get what fraction of errors are due to each component (RNN model vs Beam Search).

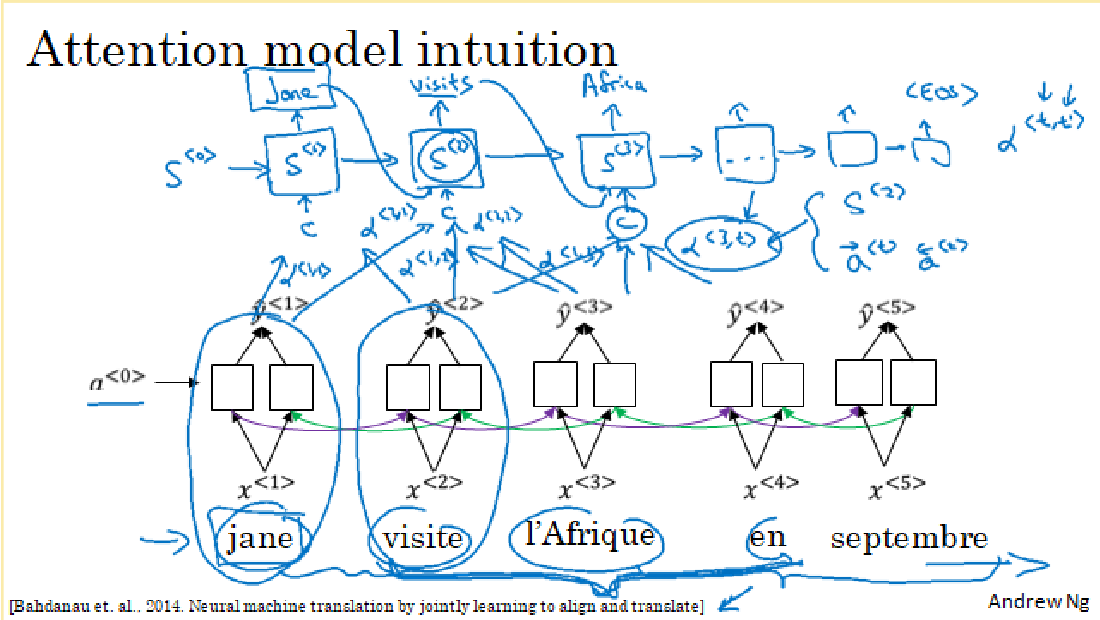
3.1.5. Bleu Score

The intuiton of the BLEU score is the following: it evaluates a machine translation comparing it to a human-made translations.

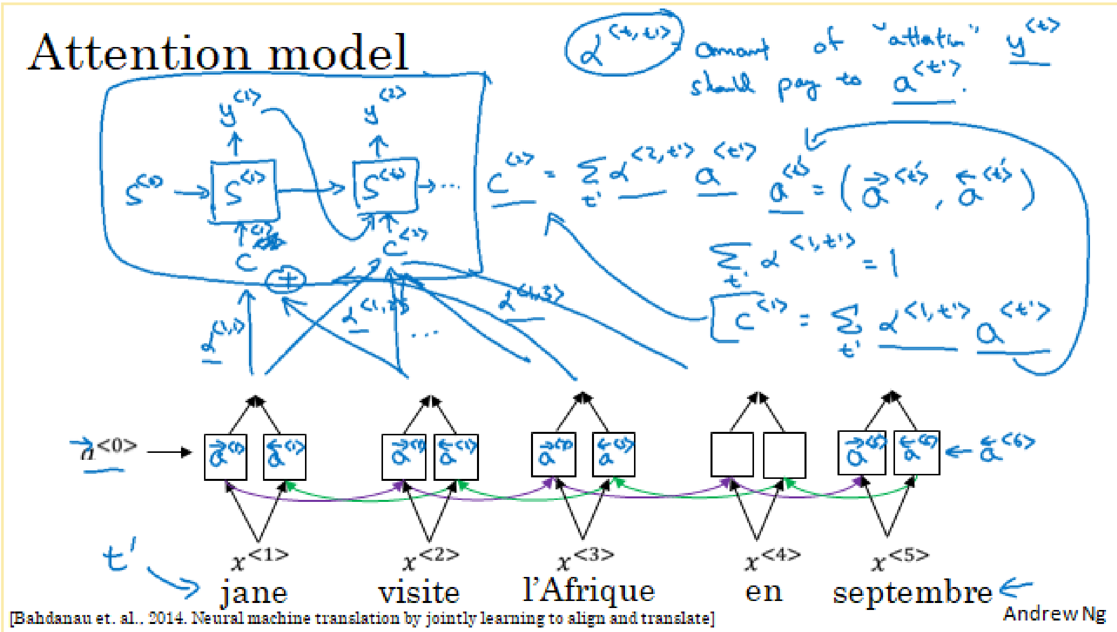
3.1.6. Attention Model

We have seen how to use an encoder-decoder architecture for machine translation. There is a modification to this called the Attention Model that makes all this work much better. This idea has been one of the most influential ideas in deep learning.

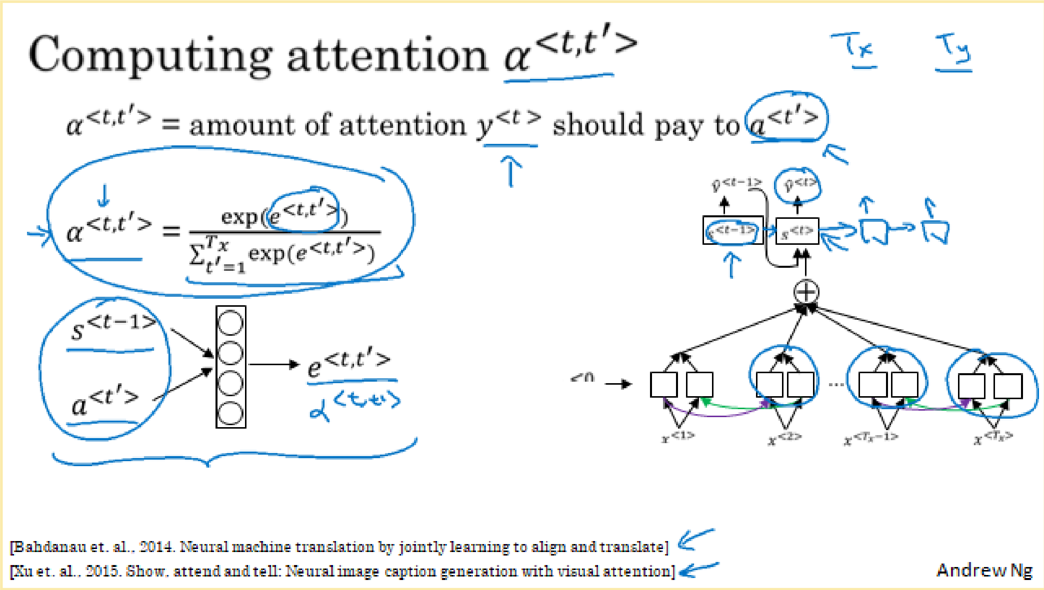
If, for example, we have a long sentence, the attention model goes part by part (just as a human would do) instead of taking the whole sentence at once. The intuition is the following: we'll compute some attention weights that will tell us to which parts of the sentence we should "pay more or less attention".



The detail of the model can be seen in the next slide:



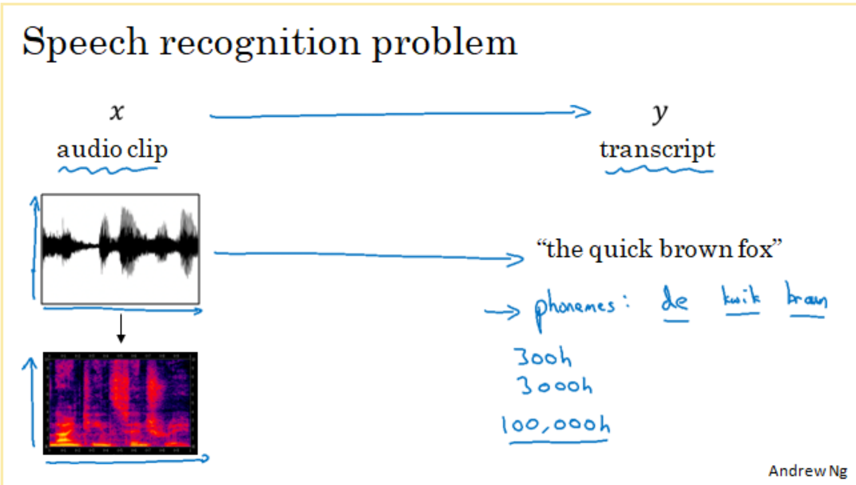
In order to compute the attention weights, we'll do the following:



3.2. Speech recognition and audio data

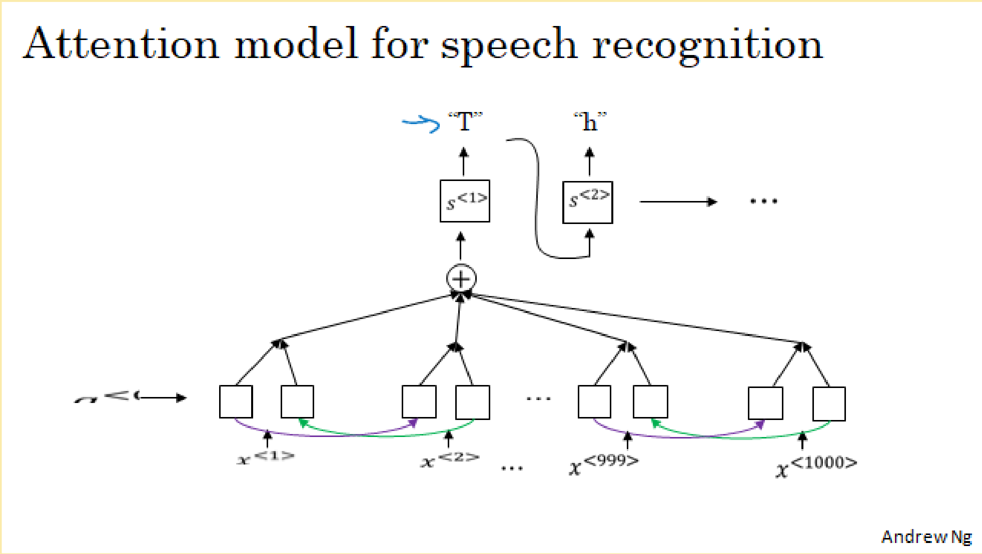
3.2.1. Speech recognition

The speech recognition problem is the following:

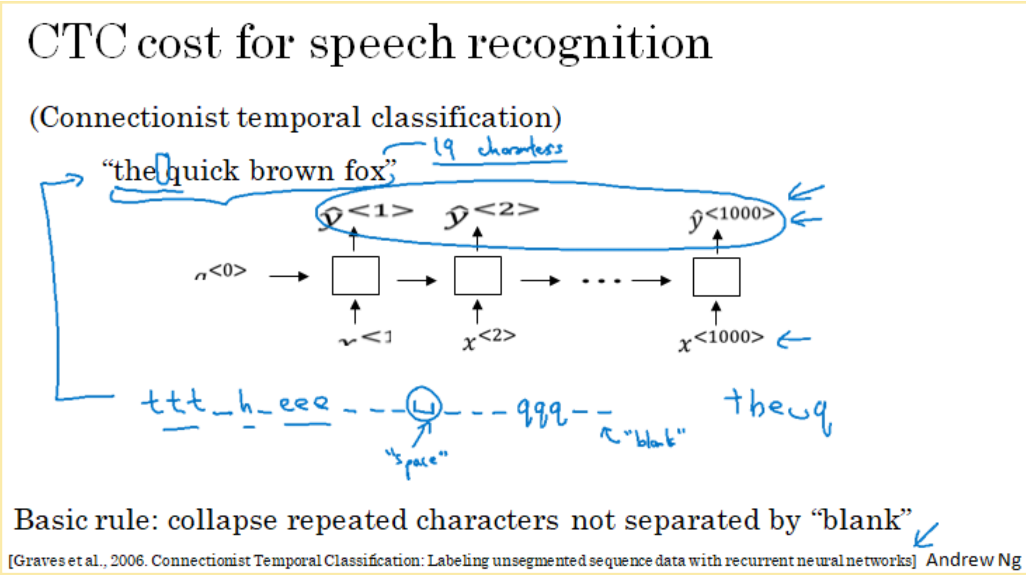


We must note that creating hand-engineered features such as phonemes are not necessary in deep learning applications.

To do speech recognition we'll build an attention model:



One other method that works well is the CTC cost for speech recognition:

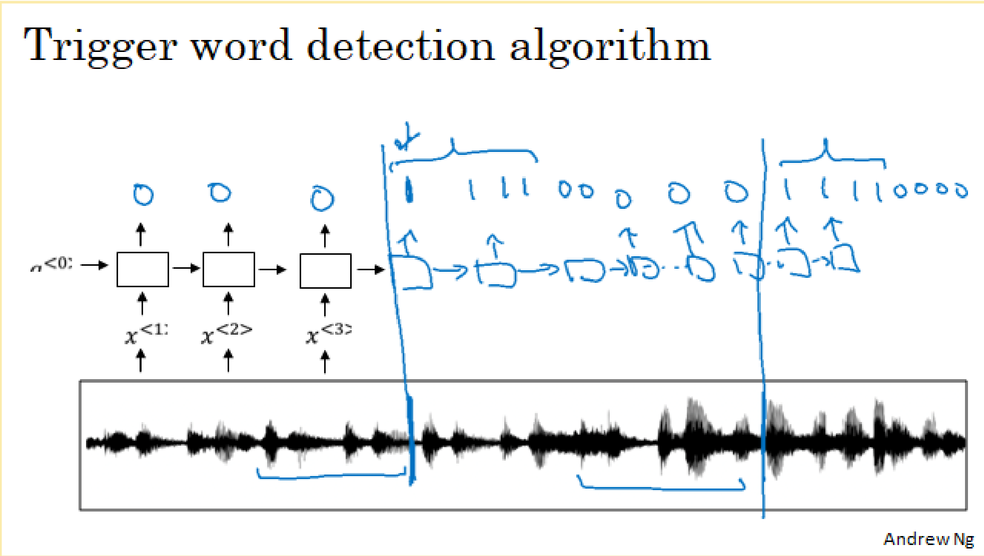


With these two approaches we can handle the speech recognition problem, but it's important to note that in order to get good results, large amounts of data are necessary. We'll see a keyword detection system, which is actually much easier and can be done with a more reasonable amount of data.

3.2.2. Trigger Word Detection

An example of trigger word detection is Siri. The system is trigged when it hears a defined word or sentence.

One example of a trigger word detection algorithm is the following:



The label will be 1 when the trigger word is pronounced.