Week 1: Recurrent Neural Networks

Recurrent neural networks have been proven to perform extremely well on temporal data. This model has several variants including LSTMs [https://en.wikipedia.org/wiki/Long_short-term_memory], GRUs [https://en.wikipedia.org/wiki/Gated_recurrent_unit] and Bidirectional RNNs [https://en.wikipedia.org/wiki/Bidirectional_recurrent_neural_networks], which you are going to learn about in this section.

Why sequence models?

Recurrent neural networks (RNNs) have proven to be incredibly powerful networks for sequence modelling tasks (where the inputs x, outputs y or both are sequences) including:

- speech recognition
- music generation
- sentiment classification
- DNA sequence analysis
- machine translation
- video activity recognition
- named entity recognition

Notation

As a motivating example, we will "build" a model that performs **named entity recognition** (**NER**).

Example input:

$$x: \text{Harry Potter and Hermione Granger invented a new spell.}$$

We want our model to output a target vector with the same number elements as our input sequence x, representing the **named entities** in x.

We will refer to each element in our input (x) and output (y) sequences with angled brackets, so for example, $x^{<1>}$ would refer to "Harry". Because we have multiple input sequences, we denote the $i-th$ sequence $x^{(i)}$ (and its corresponding output sequence $y^{(i)}$). The $t-th$ element of the $i-th$ input sequence is therefore $x^{(i)<t>}$.

Let T_x be the length of the input sequence and T_y the length of the output sequence.

> **Note**
>
> In our example, T_x == T_y

Representing words

For NLP applications, we have to decide on some way to represent words. Typically, we start by generating a **vocabulary** (a dictionary of all the words that appear in our corpus).

> **Note**

> In modern applications, a vocabulary of 30-50K is common and massive vocabularies (>1 million word types) are often used in commercial applications, especially by big tech.

A common way to represent each word is to use a **one-hot** encoding. In this way, we represent each token by a vector of dimension $||V||$ (our vocabulary size).

Example:

$$x^{<1>} = \begin{pmatrix} 0 \\ .. \\ 1 \\ ... \\ 0 \end{pmatrix}$$

$x^{<1>}$ of our sequence (i.e. the token *Harry*) is represented as a vector which contains all zeros except for a single value of one at row j, where j is its position in V.

> **Note**
>
> "One-hot" refers to the fact that each vector contains only a single 1.

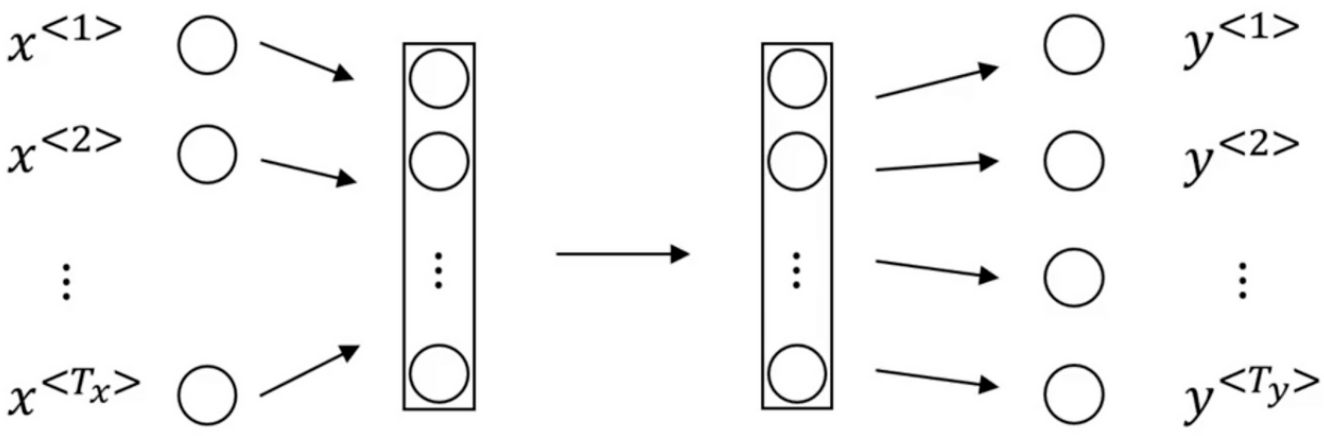
The goal is to learn a mapping from each $x^{<t>}$ to some **tag** (i.e. PERSON).

> **Note**
>
> To deal with out-of-vocabulary (OOV) tokens, we typically assign a special value *<UNK>* and a corresponding vector.

Recurrent Neural Network Model

Why not a standard network?

In our previous example, we had 9 input words. You could imagine taking these 9 input words (represented as one-hot encoded vectors) as inputs to a "standard" neural network

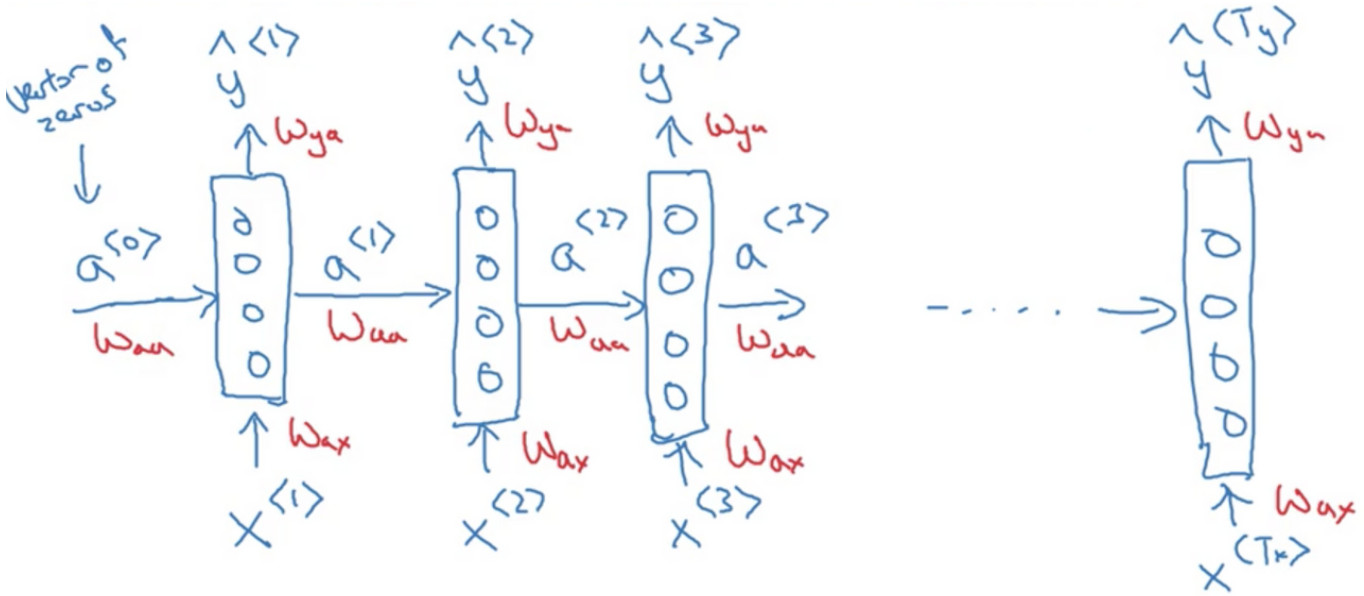


This turns out *not* to work well. There are two main problems:

1. Inputs and outputs can be different **lengths** in different examples (its not as if every T_x, T_y pair is of the same length).
2. A "standard" network doesn't **share features** learned across different positions of text. This is a problem for multiple reasons, but a big one is that this network architecture doesn't capture dependencies between elements in the sequence (e.g., the information that is a word in its context is not captured).

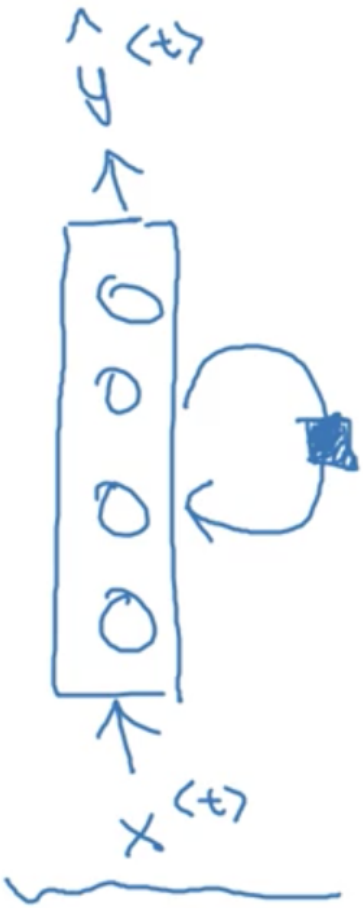
Recurrent Neural Networks

Unlike a "standard" neural network, **recurrent neural networks** (**RNN**) accept input from the *previous* timestep in a sequence. For our example x above, the *unrolled* RNN diagram might look like the following:



> **Note**
>
> Timestep 0 is usually initialized with a fake vector of 0's

Note that the diagram is sometimes drawn like this:



Where the little black box represented a delay of *1 timestep*.

A RNN learns on a sequence from *left to right*, **sharing** the parameters from **each timestep**.

- the parameters governing the connection from $x^{<t>}$ to the hidden layer will be some set of parameters we're going to write as W_{ax}.
- the activations, the horizontal connections, will be governed by some set of parameters W_{aa}
- W_{ya}, governs the output predictions

> **Note**
>
> We take the notation W_{ya}, to mean (for example) that the parameters for variable y are obtained by multiplying by some quantity a.

Notice this parameter sharing means that when we make the prediction for, $y^{<3>}$ say, the RNN gets the information not only from $x^{<3>}$ but also from the all the previous timesteps.

Note a potential weakness here. We don't incorporate information from future timesteps in our predictions. This problem is solved by using **bidirectional** RNNs (BRNNs) which we discuss in a future video.

Example:

Given the sentences:

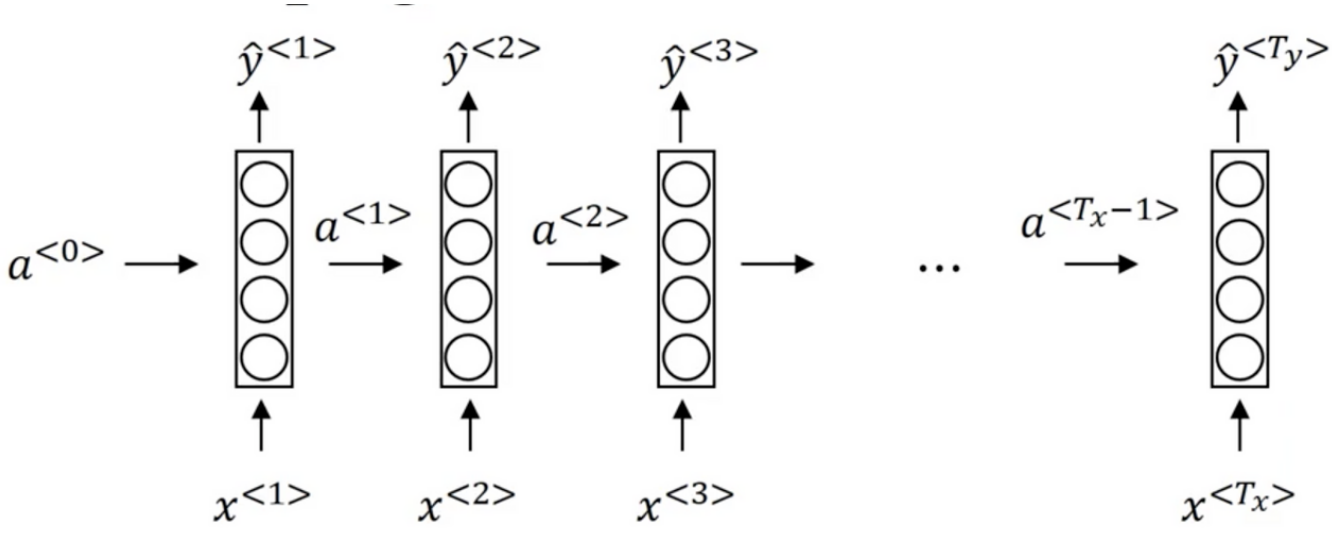
$x^{(1)}$: *He said, "Teddy Roosevelt was a great President"*

$x^{(2)}$: *He said, "Teddy bears are on sale!"*

And the task of **named entity recognition** (**NER**), it would be really useful to know that the word "*President*" follows the name "*Teddy Roosevelt*" because as the second example suggest, using only *previous* information in the sequence might not be enough to make a classification decision about an entity.

RNN Computation

Lets dig deeper into how a RNN works. First, lets start with a cleaned up depiction of our network



FORWARD PROPAGATION

Typically, we start off with the input $a^{<0>} = \vec{0}$. Then we perform our forward pass

- Compute our **activation** for timestep 1: $a^{<1>} = g(W_{aa}a^{<0>} + W_{ax}x^{<1>} + b_a)$
- Compute our **prediction** for timestep 1: $\hat{y}^{<1>} = g(W_{ya}a^{<1>} + b_y)$

More generally:

$$a^{<t>} = g(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$$

$$\hat{y}^{<t>} = g(W_{ya}a^{<t-1>} + b_y)$$

> **Note**
>
> Where b is our bias vector.

The activation function used for the units of a RNN is most commonly **tanh**, although **ReLU** is sometimes used. For the output units, it depends on our problem. Often **sigmoid** / **softmax** are used for binary and multi-class classification problems respectively.

Simplified RNN Notation

Lets take the general equations for forward propagation we developed above:

$$a^{<t>} = g(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$$

$$\hat{y}^{<t>} = g(W_{ya}a^{<t-1>} + b_y)$$

We define our simplified **hidden activation** formulation:

$$a^{<t>} = g(W_a[a^{<t-1>}, x^{<t>}] + b_a)$$

Where

$$W_a = \begin{pmatrix} W_{aa} & | & W_{ax} \end{pmatrix}$$

$$[a^{<t-1>}, x^{<t>}] = \begin{pmatrix} a^{<t-1>} \\ x^{<t>} \end{pmatrix}$$

> **Note**
>
> $W_a[a^{<t-1>}, x^{<t>}] = W_{aa}a^{<t-1>} + W_{ax}x^{<t>}$

The advantages of this notation is that we can compress two parameter matrices into one.

And our simplified **output activation** formulation:

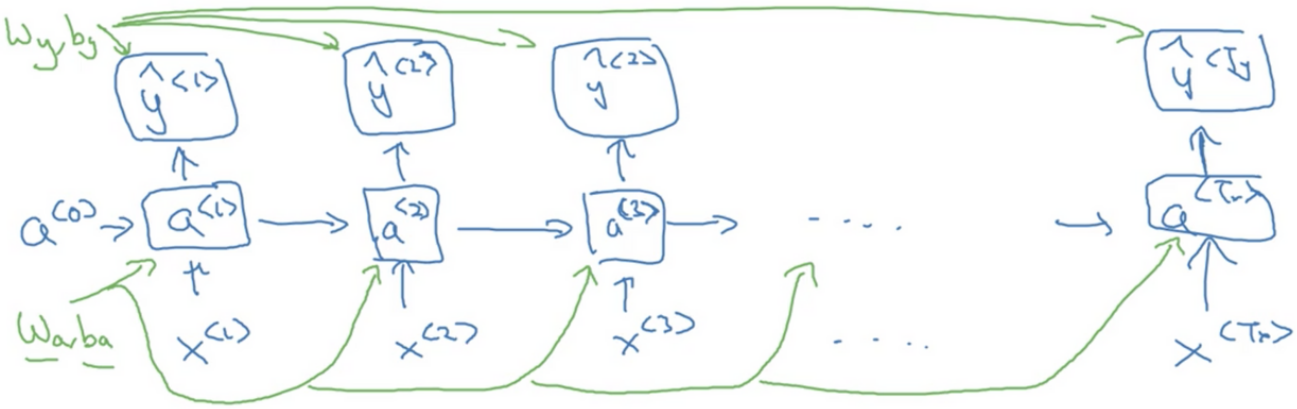
$$\hat{y}^{<t>} = g(W_y a^{<t-1>} + b_y)$$

Backpropagation through time

Forward propagation

We have seen at a high-level how forward propagation works for an RNN. Essentially, we forward propagate the input, multiplying it by our weight matrices and applying our activation for each timestep until we have outputted a prediction for each timestep in the input sequence.

More explicitly, we can represent the process of foward propogation as a series of matrix multiplications in diagram form:



Backward propogation through time (BPTT)

In order to perform **backward propagation through time** (**BPTT**), we first have to specify a loss function. We will choose **cross-entropy** loss (we also saw this when discussing logisitc regression):

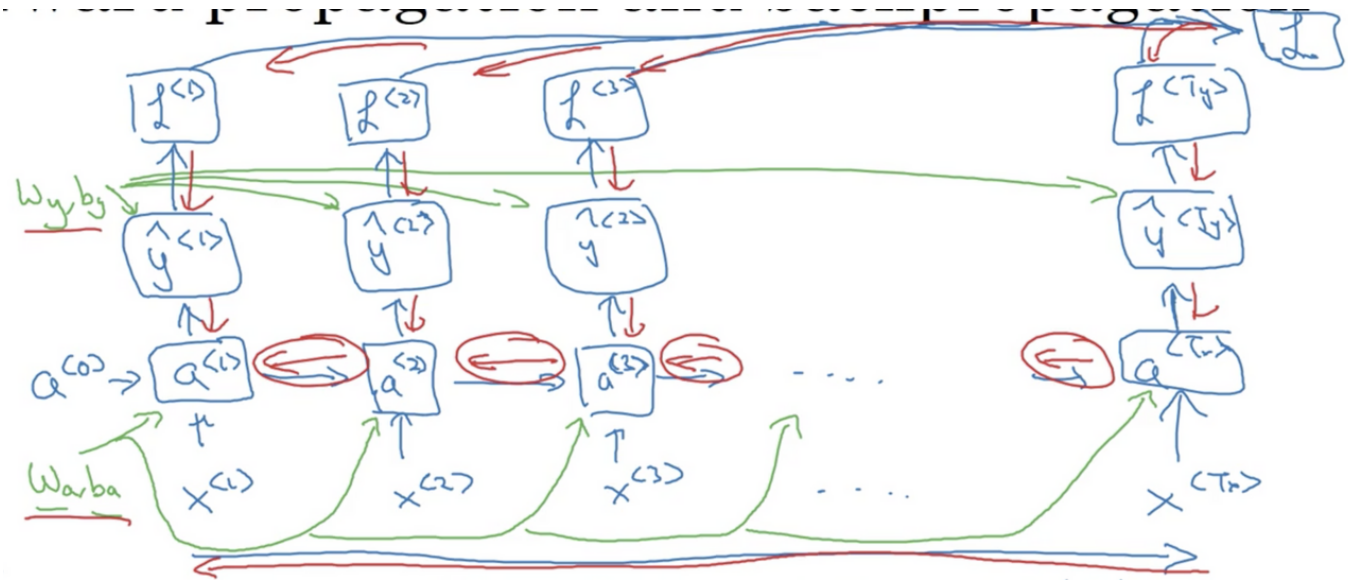
For a single prediction (timestep)

$$\ell^{<t>}(\hat{y}^{<t>}, y^{<t>}) = -y^{<t>} \log \hat{y}^{<t>} - (1 - y^{<t>}) \log(1 - \hat{y}^{<t>})$$

For all predictions

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

While not covered in detail here, BPTT simply involves applying our loss function to each prediction at each timestep, and then using this information along with the chain rule to compute the gradients we will need to update our parameters and *assign blame proportionally*. The entire process might look something like the following:



Different types of RNNs

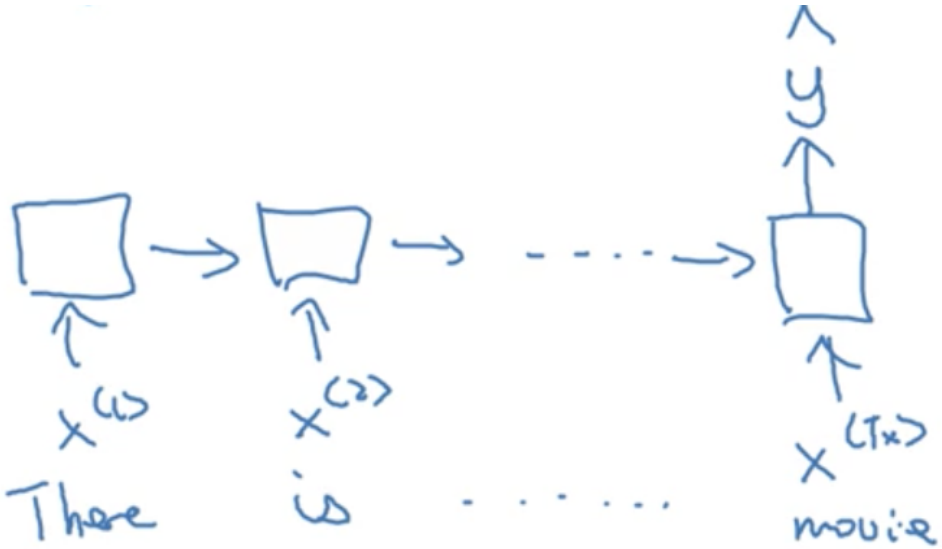
So far, we have only seen an RNN where the input and output are both sequences of lengths > 1. Particularly, our input and output sequences were of the same length ($T_x == T_y$), For many applications, $T_x \neq T_y$.

Take **sentiment classification** for example, where the input is typically a sequence (of text) and the output a integer scale (a 1-5 star review, for example).

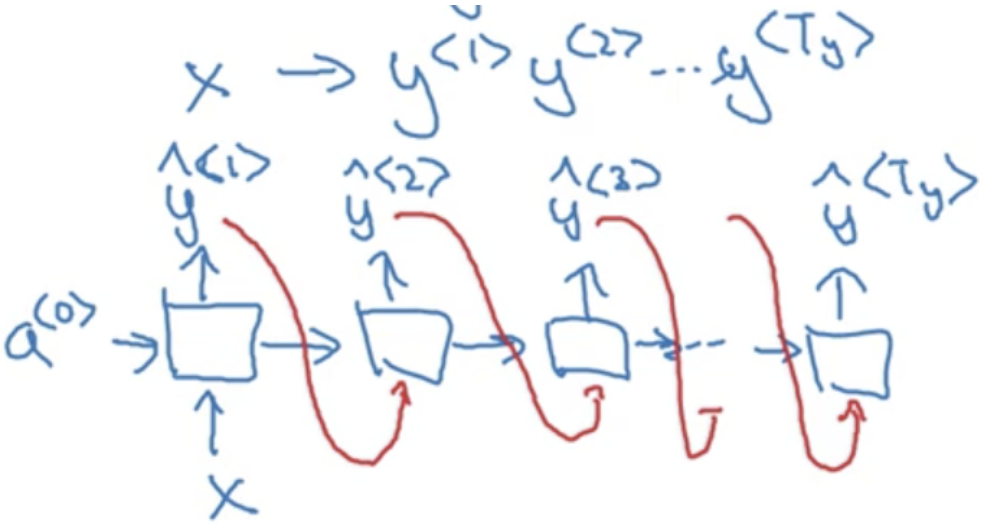
Example:

x: "There is nothing to like in this movie"

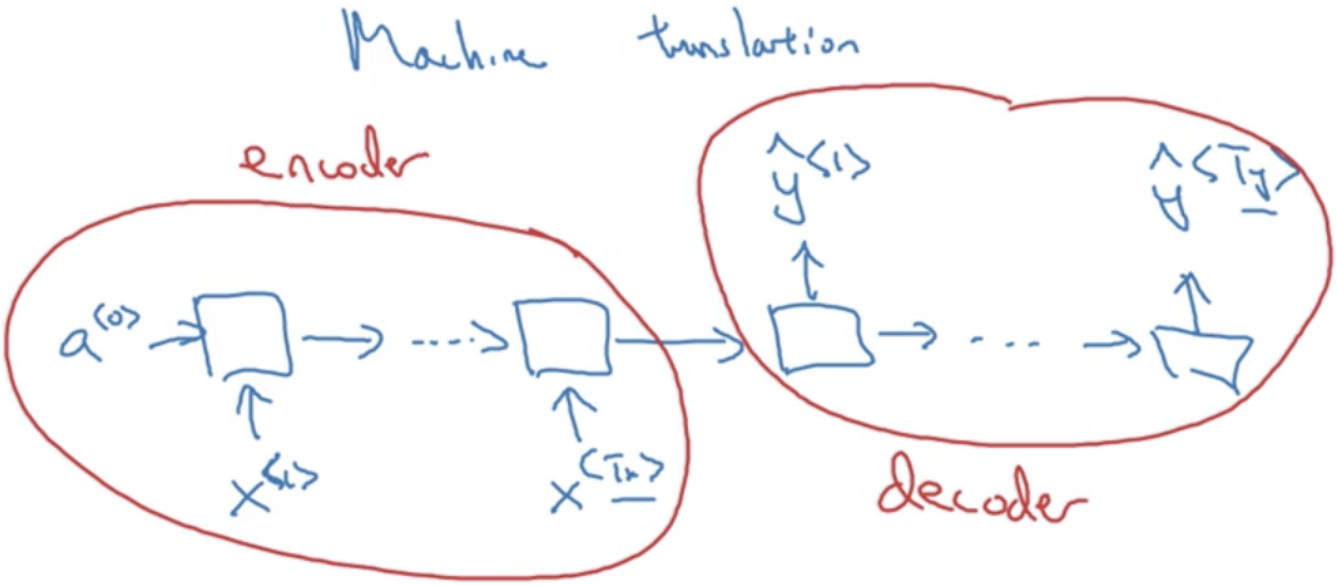
We want our network to output a single prediction from 1-5. This is an example of a **many-to-one** architecture.



Another type of RNN architecture is **one-to-many**. An example of this architecture is **music generation**, where we might input an integer (indicating a genre) or the 0-vector (no input) and generate musical notes as our output. In this case, we input a single value to the network at timestep 1, and then propagate that input through the network (the remaining timesteps), with the caveat that in this architecture, we often take the ouput from the previous timestep and feed it to the next timestep:

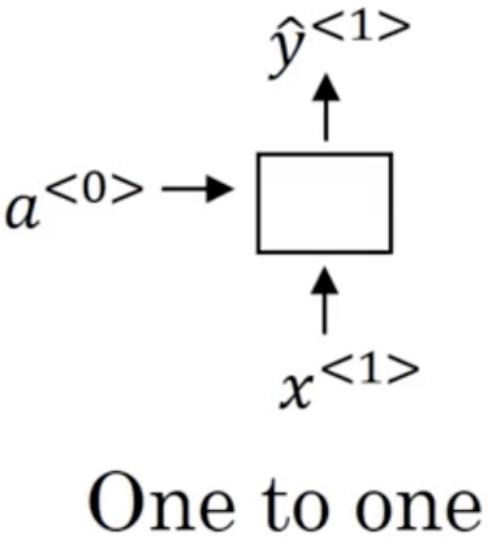


The final example is a **many-to-many** architecture. Unlike our previous example where $T_x == T_y$, in machine translation $T_x \neq T_y$, as the number of words in the input sentence (say, in *english*) is not necessarily the same as the output sentence (say, in *french*). These problems are typicaly solved with **sequence to sequence models**, that are composed of distinct **encoder** and **decoder** RNNs.



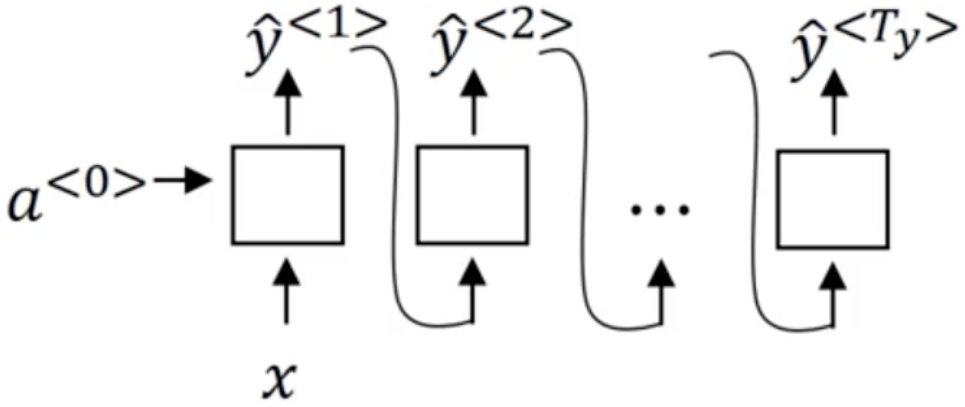
Summary of RNN types

1. **One-to-one**: a standard, generic neural network. Strictly speaking, you wouldn't model this problem with an RNN.



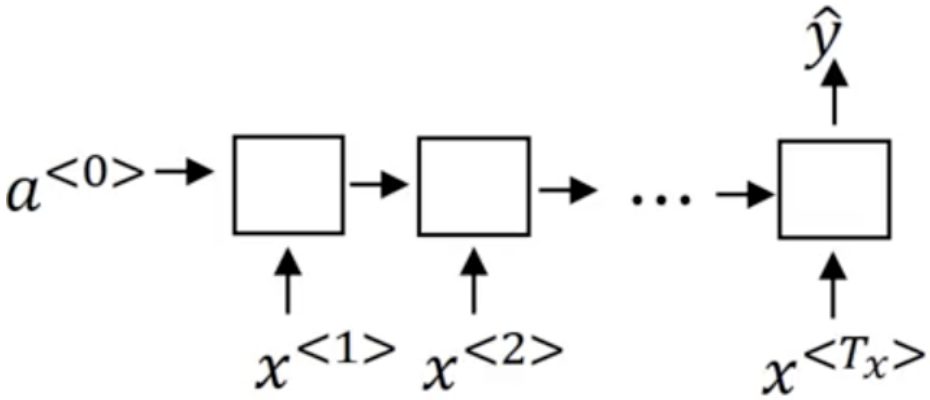
One to one

1. **One-to-many**: Where our input is a single value (or in some cases, a null input represented by the 0-vector) which propogates through the network and our output is a sequence. Often, we use the prediction from the previous timestep when computing the hidden activations. An example is *music generation* or *sequence generation* more generally.



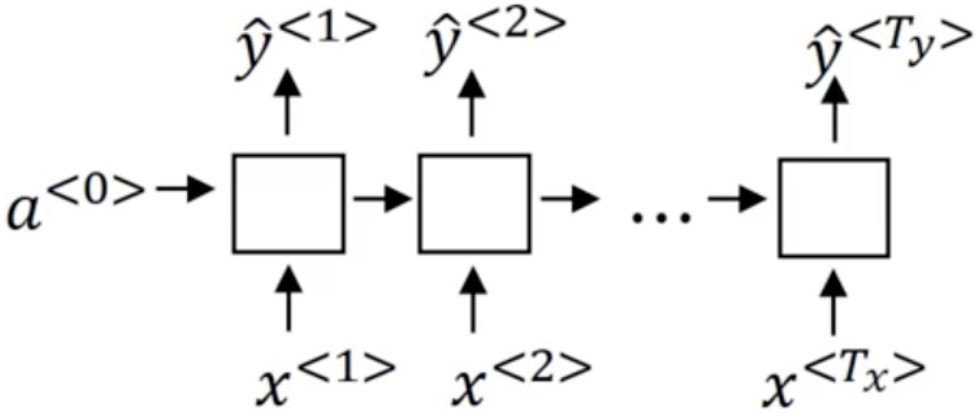
One to many

1. **Many-to-one**: Where our input is a sequence and our output is a single value. Typically we take the prediction from the last timestep of the RNN. An example is *sentiment classification*

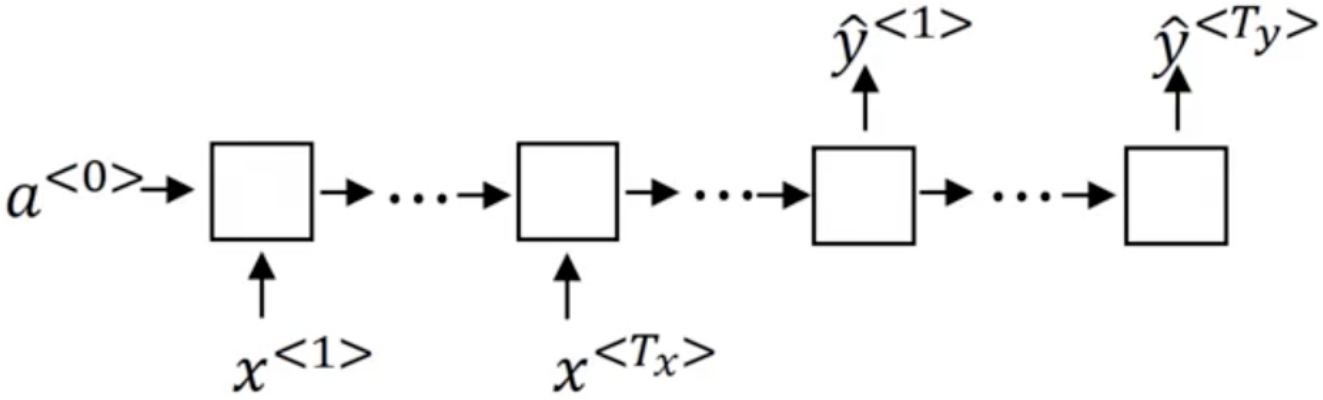


Many to one

1. **Many-to-many**: Where both our input and outputs are sequences. These sequence are not necessarily the same length ($T_x \neq T_y$).
2. When $T_x == T_y$ our architecture looks like a standard RNN:



- and when $T_x \neq T_y$ are architecture is a *sequence to sequence* model which looks like:



Language model and sequence generation

Language modeling is one of the most basic and important tasks in natural language processing. It's also one that RNNs handle very well.

What is a language modeling?

Let's say you are building a **speech recognition** system and you hear the sentence:

$$\text{"The apple and pear/pair salad"}$$

How does a neural network determine whether the speaker said *pear* or *pair* (never mind that the correct answer is obvious to us). The answer is that the network encodes a **language model**. This language model is able to determine the *probability* of a given sentence (think of this as a measure of "correctness" or "goodness"). For example, our language model might output:

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

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

This system would then pick the much more likely second option.

Language modeling with an RNN

We start with a large corpus of english text. The first step is to **tokenize** the text in order to form a vocabulary

$$\text{"Cats average 15 hours of sleep a day"} \rightarrow [\text{"Cats"}, \text{"average"}, \text{"15"}, \text{"hours"}, \text{"of"}, \text{"sleep"}, \text{"a"}, \text{"day"}, \text{"."}]$$

These tokens are then **one-hot encoded** or mapped to **indices**. Sometimes, a special end-of-sentence token is appended to each sequence (*<EOS>*).

> **Note**
>
> What if some of the words we encounter are not in our vocabulary? Typically we add a special token, *<UNK>* to deal with this problem.

Finally, we build an RNN to model the likelihood of any given sentence, learned from the training corpus.

RNN model

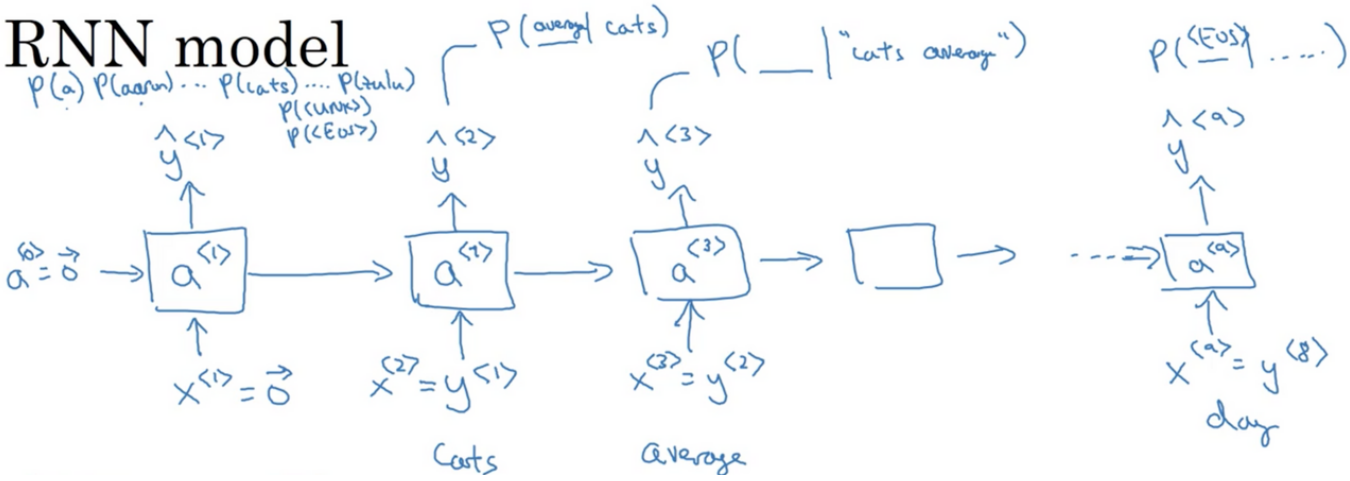
At time 0, we compute some activation $a^{<1>}$ as a function of some inputs $x^{<1>}$. In this case, $x^{<1>}$ will just be set to the zero vector. Similarly, $a^{<0>}$, by convention, is also set to the zero vector.

$a^{<1>}$ will make a *softmax* prediction over the entire vocabulary to determine $\hat{y}^{<1>}$ (the probability of observing any of the tokens in your vocabulary as the *first* word in a sentence).

At the second timestep, we will actually feed the first token in the sequence as the input ($x^{<2>} = y^{<1>}$). This occurs, so forth and so on, such that the input to each timestep are the tokens for all previous timesteps. Our outputs $\hat{y}^{<t>}$ are therefore $P(x^{<t>}|x^{<t-1>}, x^{<t-2>}, \dots, x^{<t-n>})$ where n is the length of the sequence.

> **Note**
>
> Just a note here, we are choosing $x^{<t>} = y^{<t-1>}$ *NOT* $x^{<t>} = \hat{y}^{<t-1>}$

The full model looks something like:



[https://postimg.cc/image/sqgm18ty7/]

There are two important steps in this process:

1. Estimate $\hat{y}^{<t>} = P(y^{<t>}|y^{<1>}, y^{<2>}, \dots, y^{<t-1>})$
2. Then pass the ground-truth word from the training set to the next time-step.

The **loss function** is simply the **cross-entropy** lost function that we saw earlier:

- For single examples: $\ell(\hat{y}^{<t>}, y^{<t>}) = -\sum_i y_i^{<t>} \log \hat{y}_i^{<t>}$
- For the entire training set: $\ell = \sum_i \ell^{<t>}(\hat{y}^{<t>}, y^{<t>})$

Once trained, the RNN will be able to predict the probability of any given sentence (we simply multiply the probabilities output by the RNN at each timestep).

Sampling novel sequences

After you train a sequence model, one of the ways you can get an informal sense of what is learned is to sample novel sequences (also known as an *intrinsic evaluation*). Let's take a look at how you could do that.

Word-level models

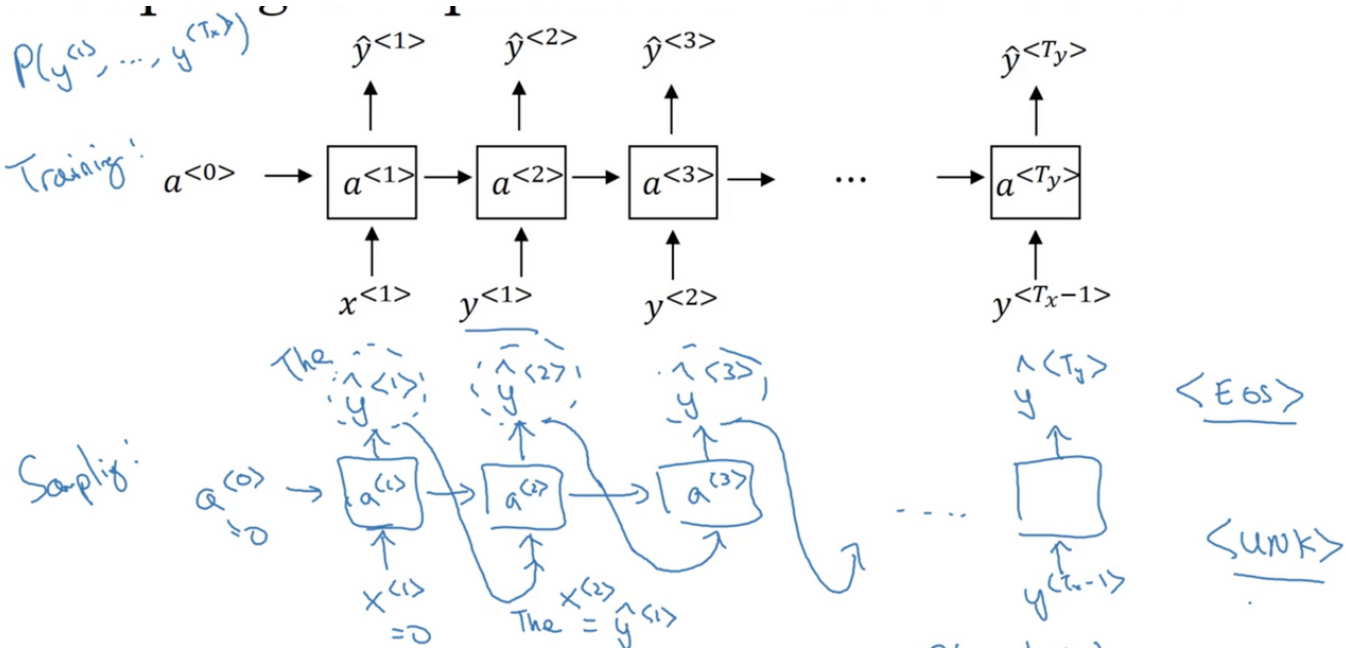
Remember that a sequence model models the probability of any given sequence of words. What we would like to to is to *sample* from this distribution to generate *novel* sequence of words.

> **Note**
>
> At this point, Andrew makes a distinction between the architecture used for *training* a language modeling and the architecture used for *sampling* from a language model. The distinction is completely lost on me.

We start by computing the activation $a^{<1>}$ as a function of some inputs $x^{<1>}$ and $a^{<0>}$ (again, these are set to the zero vector by convention). The **softmax** function is used to generate a probability distribution over all words in the vocabulary, representing the likelihood of seeing each at the first position of a word sequence. We then randomly sample from this distribution, choosing a single token ($\hat{y}^{<1>}$), and pass it as input for the next timestep.

> **Note**
>
> For example, if we sampled "the" in the first timestep, we would set $\hat{y}^{<1>} = the = x^{<2>}$. This means that at the second timestep, we are computing a probability distribution $P(v|the)$ over all tokens v in our vocabulary V.

The entire procedure looks something like:



[https://postimg.cc/image/cf7rww47z/]

How do we know when the sequence ends?

If we included the token in our training procedure (and this included it in our vocabulary) the sequence ends when and <EOS> token is generated. Otherwise, stop when a pre-determined number of tokens has been reached.

What if we generate an <UNK> token?

We can simply re-sample until we generate a non-<UNK> token.

Character-level models

We could also build a **character-level language model**. The only major difference is that we train on a sequence of *characters* as opposed to *tokens*, and therefore our vocabulary consists of individual *characters* (which typically include digits, punctuation, etc.)

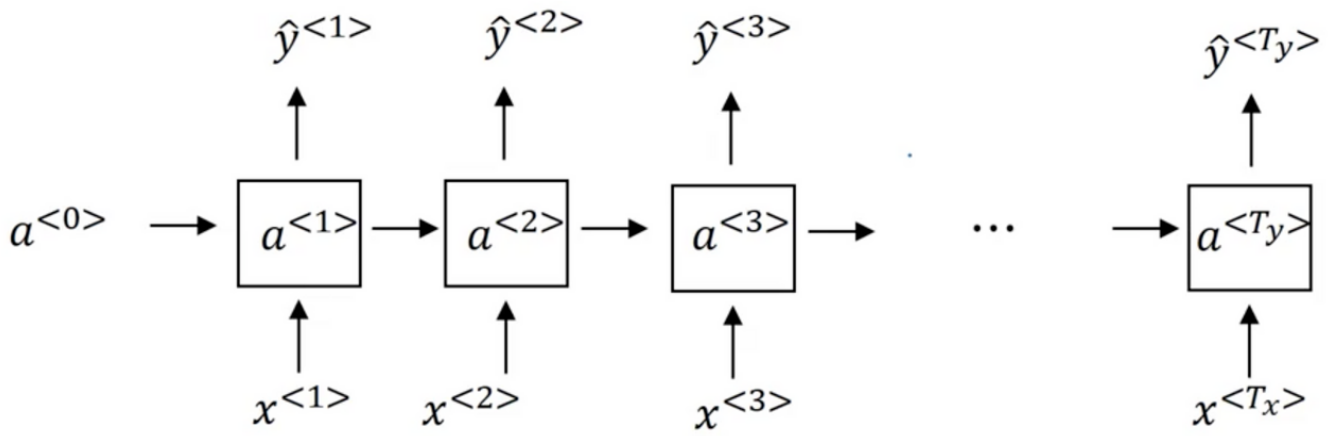
Character-level language models are more computational expensive, and because a sequence of characters is typically much longer than a sequence of words (obviously) it is more difficult to capture the long range dependencies (as they are longer, of course).

However, using a *character*-level language models has the benefit of avoiding the problem of out-of-vocabulary tokens, as we can build a non-zero vector representation of *any* token using the learned character representations.

> **Note**
>
> You can also combine word-level and character-level language models!

Vanishing gradients with RNNs

One of the problems with the basic RNN algorithm is the **vanishing gradient problem**. The RNN architecture as we have described it so far:



[https://postimg.cc/image/th0lz2be7/]

Take the following two input examples:

$$x^{(1)} = \text{The cat, which already ate ..., was full}$$

$$x^{(2)} = \text{The cats, which already ate ..., were full}$$

> **Note**
>
> Take the "..." to be an sequence of english words of arbitrary length.

Cleary, there is a long range-dependency here between the *grammatical number* of the **noun** "cat" and the *grammatical tense* of the **verb** "was".

> **Note**
>
> It is important to note that while this is a contrived example, language very often contains long-range dependencies.

It turns out that the basic RNNs that we have described thus far is not good at capturing such long-range dependencies. To explain why, think back to our earlier discussions about the *vanishing gradient problems* in very deep neural networks. The basic idea is that in a network with many layers, the gradient becomes increasingly smaller as it is backpropagated through a very deep network, effectively "vanishing". RNNs face the same problem, leading to errors in the outputs of later timesteps having little effect on the gradients of earlier timesteps. This leads to a failure to capture long-range dependencies.

> **Note**
>
> Because of this problem, a basic RNN captures mainly local influences.

Recall that *exploding gradients* are a similar yet opposite problem. It turns out that *vanishing gradients* are a bigger problems for RNNs, but *exploding gradients* do occur. However, *exploding gradients* are typically easier to catch as we simply need to look for gradients that become very very large (also, they usually lead to computational overflow, and generate NaNs). The solution to *exploding gradient* problems is fairly straightforward however, as we can use a technique like **gradient clipping** to scale our gradients according to some maximum values.

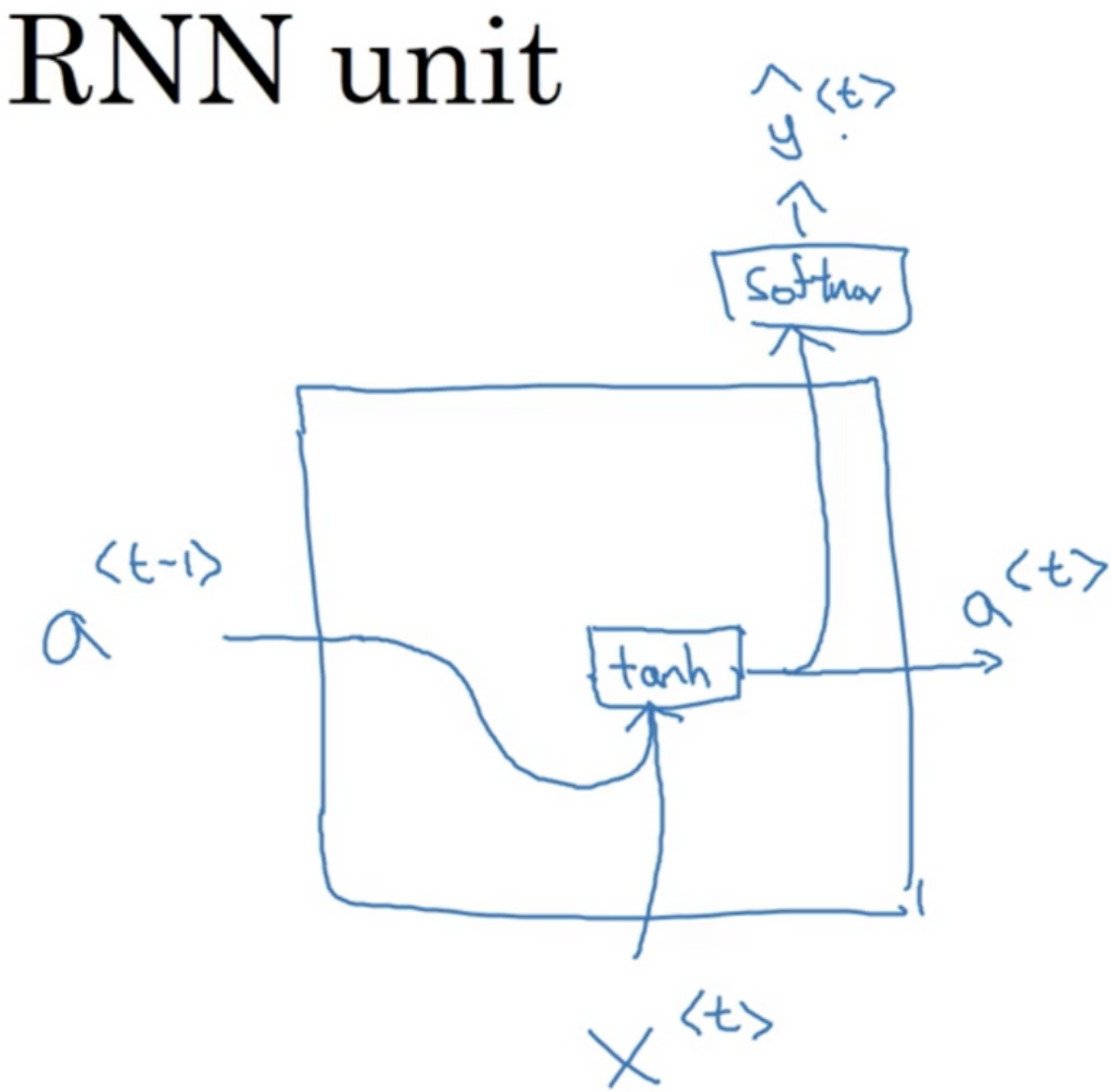
Gated Recurrent Unit (GRU)

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

Recall the activation function for an RNN at timestep t:

$$a^{<t>} = g(W_a[a^{<t-1>}, x^{<t>}] + b_a)$$

As a picture:



[https://postimg.cc/image/ceczul8fj/]

> **Note**
>
> Two papers were important for the development of GRUs: Cho et al., 2014 [https://arxiv.org/pdf/1406.1078v3.pdf] and Chung et al., 2014 [https://arxiv.org/pdf/1412.3555.pdf].

Lets define a new variable, c for the **memory cell**. The job of the memory cell is to remember information earlier in a sequence. So at time $< t >$ the memory cell will have some value $c^{<t>}$. In GRUs, it turns out that $c^{<t>} == a^{<t>}$.

> **Note**
>
> It will be useful to use the distinct variables however, as in LSTM networks $c^{<t>} \neq a^{<t>}$

At every timestep t, we are going to consider overwriting the value of the memory cell $c^{<t>}$ with a new value, computed with an activation function:

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

The most important idea in the GRU is that of an **update gate**, Γ_u, which always has a value between 0 and 1:

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

> **Note**
>
> Subscript u stands for update.

To build our intuition, think about the example we introduced earlier:

$$x^{(1)} = \text{The cat, which already ate ..., was full}$$

We noted that here, the fact that the word *"cat"* was singular was a huge hint that the verb *"was"* would also be singular in number. We can imagine $c^{<t>}$ as *memorizing* the case of the noun "*cat*" until it reached the verb "*was*". The job of the **gate** would be to *remember* this information between "... *cat* ... *were* ..." and *forget* it afterwords.

To compute $c^{<t>}$:

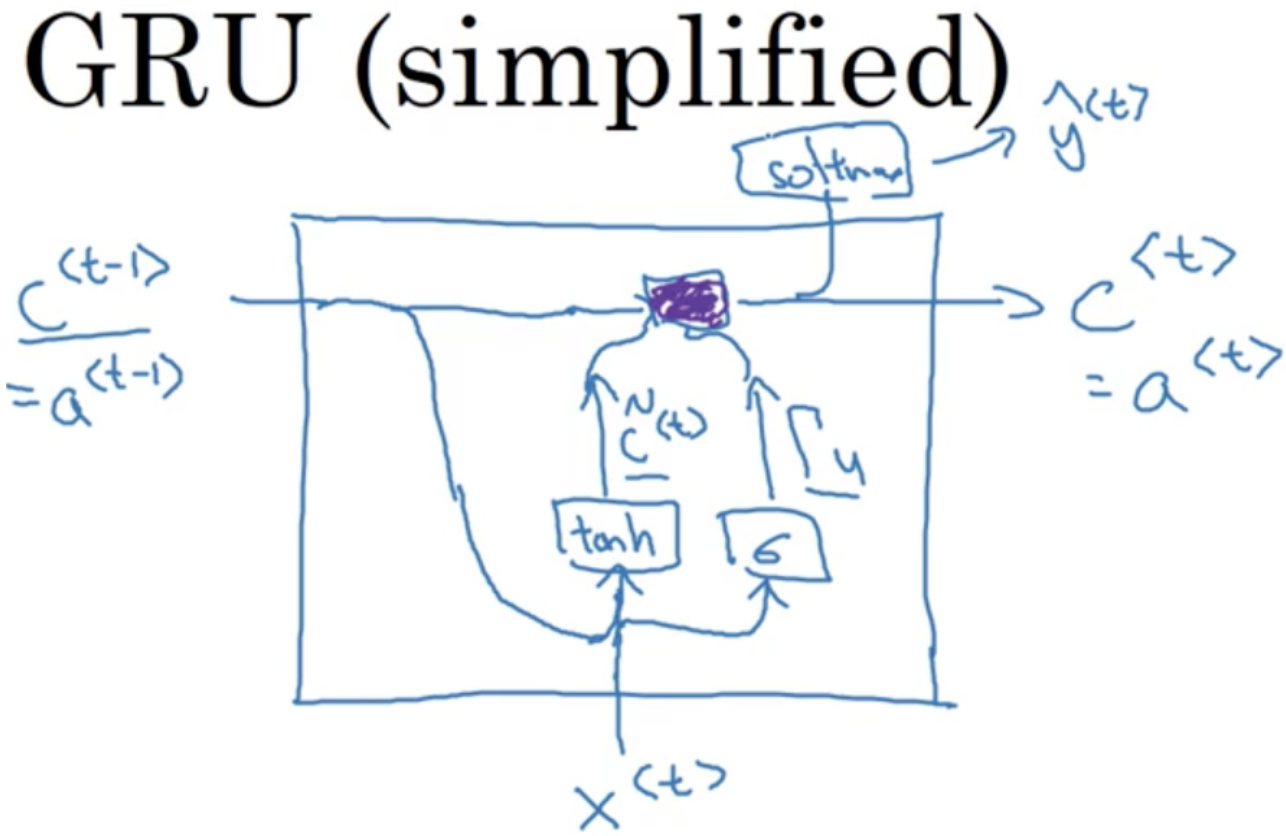
$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t>}$$

There is a very intuitive understanding of this computation. When Γ_u is 1, we simply *forget* the old value of $c^{<t>}$ by overwriting it with $\tilde{c}^{<t>}$. When Γ_u is 0, we do the opposite (completely diregard the new candidate memory $\tilde{c}^{<t>}$ in favour of the old memory cell value $c^{<t>}$).

> **Note**
>
> Remember that Γ_u can take on any value between 0 and 1. The larger the value, the more weight that the candidate memory cell value takes over the old memory cell value.

For our example sentence above, we might hope that the GRU would set $\Gamma_u = 1$ once it reached "*cats*", and then $\Gamma_u = 0$ for every other timestep until it reached "*was*", where it might set $\Gamma_u = 1$ again. Think of this as the network memorizing the grammatical number of the **subject** of the sentence in order to determine the number of its verb, a concept known as **agreement**.

As a picture:



[https://postimg.cc/image/v83e5cj6n/]

> **Note**
>
> The purple box just represents our calculation of $c^{<t>}$

GRUs are remarkably good at determining when to update the memory cell in order to **memorize** or **forget** information in the sequence.

Vanishing gradient problem

The way a GRU solves the vanishing gradient problem is straightforward: the **memory cell** $c^{<t>}$ is able to retain information over many timesteps. Even if Γ_u becomes very very small, $c^{<t>}$ will essentially retain its value across many many timesteps.

Implementation details

$c^{<t>}, \tilde{c}^{<t>}$ and Γ_u are all vectors of the same dimension. This means that in the computation of:

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

$*$ are *element-wise* multiplications. Thus, if Γ_u is a 100-dimensional vector, it is really a 100-dimensional vector of *bits* which tells us of the 100-dimensional memory cell $c^{<t>}$, which are the *bits* we want to *update*.

> **Note**
>
> Of course, in practice Γ_u will take on values that are not *exactly* 0 or 1, but its helpful to image it as a bit vector to build our intuition.

Invoking our earlier example one more time:

$$x^{(1)} = \text{The cat, which already ate ..., was full}$$

we could imagine representing the grammatical number of the noun "*cat*" as a single *bit* in the memory cell.

Full GRU unit

The description of a GRU unit provided above is actually somewhat simplified. Below is the computations for the *full* GRU unit:

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

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

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

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

We introduce another gate, Γ_r. Where we can think of this gate as capturing how relevant $c^{<t-1>}$ is for computing the next candidate $c^{<t>}$.

> **Note**
>
> You can think of r as standing for relevance.

Note that Andrew tried to establish a consistent notation to use for explaining both GRUs and LSTMs. In the academic literature, you might often see:

- $\tilde{c}^{<t>} : \tilde{h}$
- $\Gamma_u : u$
- $\Gamma_r : r$
- $c^{<t>} : h$

> **Note**
>
> (our notation : common academic notation)

Long Short Term Memory (LSTM)

In the last video, you learned about the **GRU**, and how that can allow you to learn very long range dependencies in a sequence. The other type of unit that allows you to do this very well is the **LSTM** or the **long short term memory** units, and is even more powerful than the GRU.

Recall the full set of equations defining a GRU above:

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

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

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

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

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

The LSTM unit is a more powerful and slightly more general version of the GRU (in truth, the LSTM was defined before the GRU). Its computations are defined as follows:

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

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

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

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

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

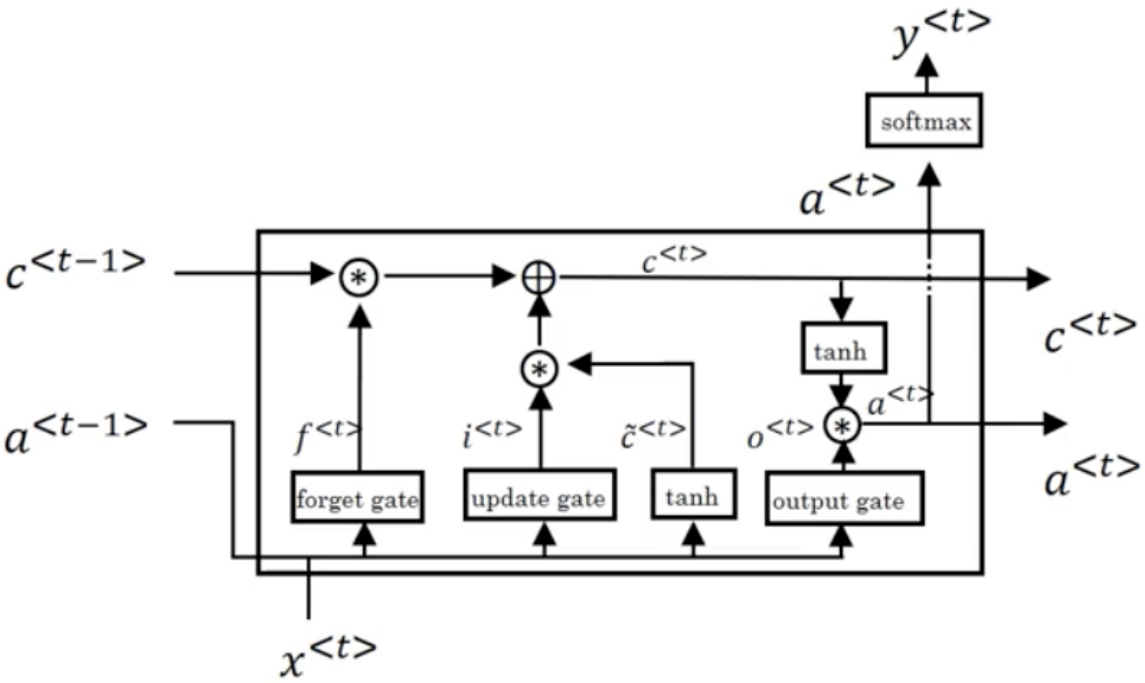
$$a^{<t>} = \Gamma_o * tanh(c^{<t>})$$

Note

Original LSTM paper [https://dl.acm.org/citation.cfm?id=1246450].

Notice that with LSTMs, $a^{<t>} \neq c^{<t>}$. One new property of the LSTM is that instead of *one* update gate, Γ_u, we have *two* update gates, Γ_u and Γ_f (for **update** and **forget** respectively). This gives the memory cell the option of keeping the old memory cell information $c^{<t-1>}$ and just adding to it some new information $\tilde{c}^{<t>}$.

We can represent the LSTM unit in diagram form as follows:

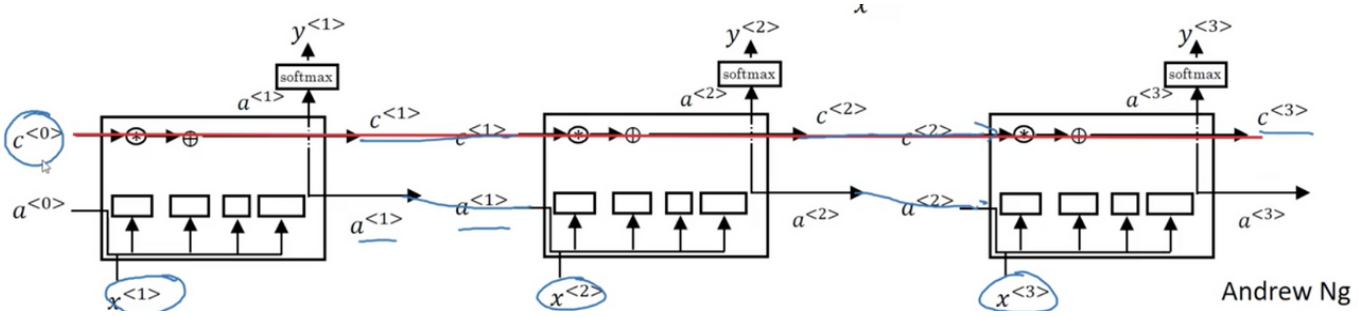


[https://postimg.cc/image/f6gixd127/]

Note

See here [https://colah.github.io/posts/2015-08-Understanding-LSTMs/] for an more detailed explanation of an LSTM unit.

One thing you may notice is that if we draw out multiple units in temporal succession, it becomes clear how the LSTM is able to achieve something akin to "memory" over a sequence:



[https://postimg.cc/image/mz76pcer3/]

Modifications to LSTMs

There are many modifications to the LSTM described above. One involves including $c^{<t-1>}$ along with $a^{<t-1>}, x^{<t>}$ in the gate computations, known as a **peephole connection**. This allows for the gate values to depend not just on the input and the previous timesteps activation, but also on the previous timesteps value of the memory cell.

Bidirectional RNNs (BRNNs)

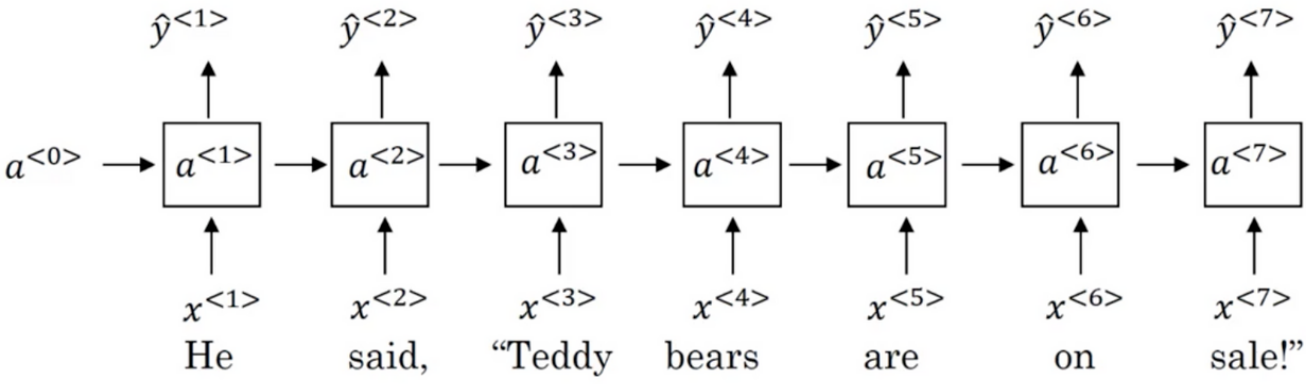
By now, you've seen most of the building blocks of RNNs. There are two more ideas that let you build much more powerful models. One is **bidirectional RNNs** (**BRNNs**), which lets you at a point in time to take information from both earlier and later in the sequence. And second, is deep RNNs, which you'll see in the next video.

To motivate bidirectional RNNs, we will look at an example we saw previously:

$x^{(1)}$: *He said, "Teddy Roosevelt was a great President"*

$x^{(2)}$: *He said, "Teddy bears are on sale!"*

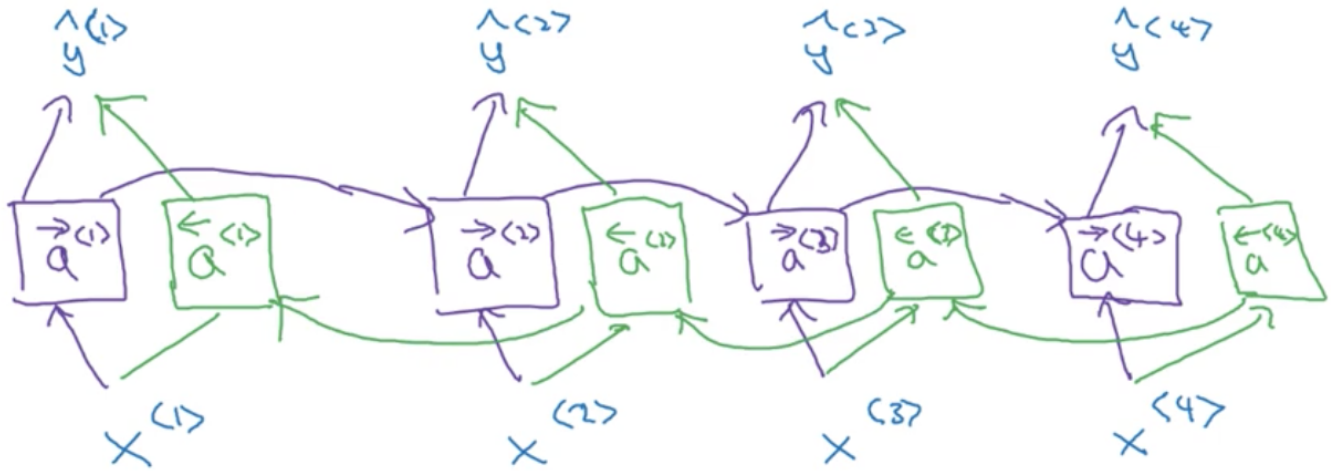
Recall, that for the task of NER we established that correctly predicting the token *Teddy* as a *person* entity without seeing the words that follow it would be difficult.



[https://postimg.cc/image/m0vf0q67j/]

> **Note**
>
> Note: this problem is independent of whether these are standard RNN, GRU, or LSTM units.

A solution to this problem is to introduce another RNN in the opposite direction, going *backwards* in time.



[https://postimg.cc/image/o5fs1t04f/]

During forward propogation, we compute activations as we have seen previously, with key difference being that we learn two series of activations: one from *left-to-right* $\overrightarrow{a}^{<t>}$ and one from *right-to-left* $\overleftarrow{a}^{<t>}$. What this allows us to do is learn the representation of each element in the sequence *within its context*. Explicitly, this is done by using the output of both the forward and backward units at each time step in order to make a prediction $\hat{y}^{<t>}$:

$$\hat{y}^{<t>} = g(W_y[\overrightarrow{a}^{<t>}, \overleftarrow{a}^{<t>}] + b_y)$$

For the example given above, this means that our prediction for the token *Teddy*, $y^{<3>}$, is able to makes use of information seen previously in the sequence ($t = 3, 2, \ldots$) and future information in the sequence ($t = 4, 5, \ldots$)

> **Note**
>
> Note again that we can build bidirectional networks with standard RNN, GRU and LSTM units. Bidirectional LSTMs are extremely common.

The disadvantage of BRNNs is that we need to see the *entire* sequence before we can make any predictions. This can be a problem in applications such as real-time speech recognition.

> **Note**
>
> the BRNN will let you take into account the entire speech utterance but if you use our straightforward implementation, you need to wait for the person to stop talking to get the entire utterance before you can actually process it and make a speech recognition prediction

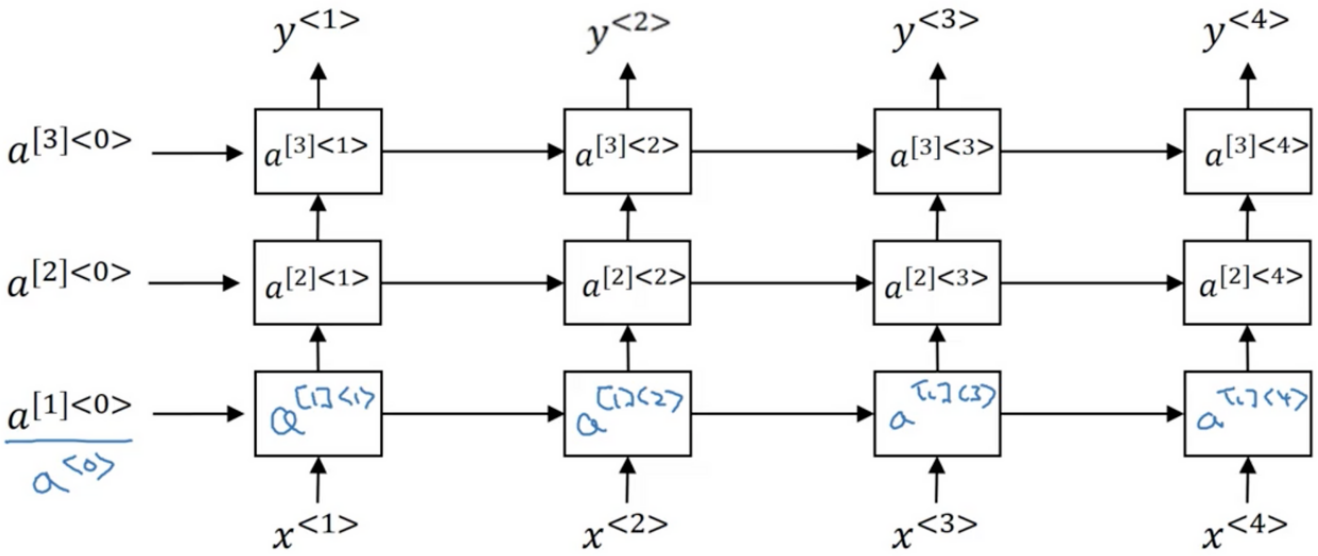
For applications like these, there exists somewhat more complex modules that allow predictions to be made before the full sequence has been seen. bidirectional RNN as you've seen here. For many NLP applications where you can get the entire sentence all the same time, our standard BRNN algorithm is actually very effective.

Deep RNNs

The different versions of RNNs you've seen so far will already work quite well by themselves. But for learning very complex functions sometimes is useful to stack multiple layers of RNNs together to build even deeper versions of these models.

Recall, that for a standard neural network we have some input x which is fed to a hidden layer with activations $a^{[l]}$ which are in turn fed to the next layer to produce activations $a^{[l+1]}$. In this was, we can stack as many layers as we like. The same is true of RNNs. Lets use the notation $a^{[l]<t>}$ to denote the activations of layer l for timestep t.

A stacked RNN would thus look something like the following:



[https://postimg.cc/image/mg6otn4yn/]

The computation of, for example, $a^{[2]<3>}$ would be:

$$a^{[2]<3>} = g(W_a^{[2]}[a^{[2]<2>}, a^{[1]<3>}] + b_a^{[2]})$$

Notice that the second layer has parameters $W_a^{[2]}$ and $b_a^{[2]}$ which are shared across all timesteps, but *not* across the layers (which have their own corresponding set of parameters).

Unlike standard neural networks, we rarely stack RNNs very deep. Part of the reason is that RNNs are already quite large due to their temporal dimension.

> **Note**
>
> A common depth would be 2 stacked RNNs.

Something that has become more common is to apply deep neural networks to the output of each timestep. In this approach, the *same* deep neural network is typically applied to each output of the final RNN layer.

Week 2: Natural Language Processing & Word Embeddings

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

Introduction to word embeddings: Word Representation

Last week, we learned about RNNs, GRUs, and LSTMs. In this week, you see how many of these ideas can be applied to **Natural Language Processing** (**NLP**), which is one of the areas of AI being revolutionized by deep learning. One of the key ideas you learn about is **word embeddings**, which is a way of representing words.

So far, we have been representing words with a vocabulary, V, of one-hot-encoded vectors. Lets quickly introduce a new notation. If the token "*Man*" is in position 5391 in our vocabulary V then we denote the corresponding one-hot-encoded vector as O_{5391}.

One of the weaknesses of this representation is that it treats each word as a "thing" onto itself, and doesn't allow a language model to generalize between words. Take the following examples:

$$x_1 : \text{"I want a glass of orange juice"}$$

$$x_2 : \text{"I want a glass of apple juice"}$$

Cleary, the example sentences are extremely semantically similar. However, in a one-hot encoding scheme, a model which has learned that x_1 is a likely sentence is unable to fully generalize to example x_2, as the relationship between "*apple*" and "*orange*" is not any closer than the relationship between "*orange*" and any other word in the vocabulary.

Notice, in fact, that the inner product [http://www.wikiwand.com/en/Dot_product] between any two one-hot encoded vectors:

$$O_i \times O_j = \vec{0} \text{ for } \forall i, j$$

And similarly, the euclidean distance [http://www.wikiwand.com/en/Euclidean_distance] between any two one-hot encoded vectors is identical:

$$||O_i - O_j|| = \sqrt{|V|} \text{ for } \forall i, j$$

To build our intuition of word embeddings, image a contrived example where we represent each word with some **feature representation**:

	Man (5391)	Woman (9853)	King (4914)	Queen (7157)	Apple (456)	Orange (6257)
Gender	-1	1	-0.95	0.97	0.00	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.00
Age	0.03	0.02	0.7	0.69	0.03	-0.02
Food	0.04	0.01	0.02	0.01	0.95	0.97
⋮ size cost a						

[https://postimg.cc/image/5mcil7jcf/]

We could imagine many features (with values -1 to 1, say) that can be used to build up a featue representation, an f_n-dimensional vector, of each word. Similarly to our one-hot representations, lets introduce a new notation e_i to represent the *embedding* of token i in our vocabulary V.

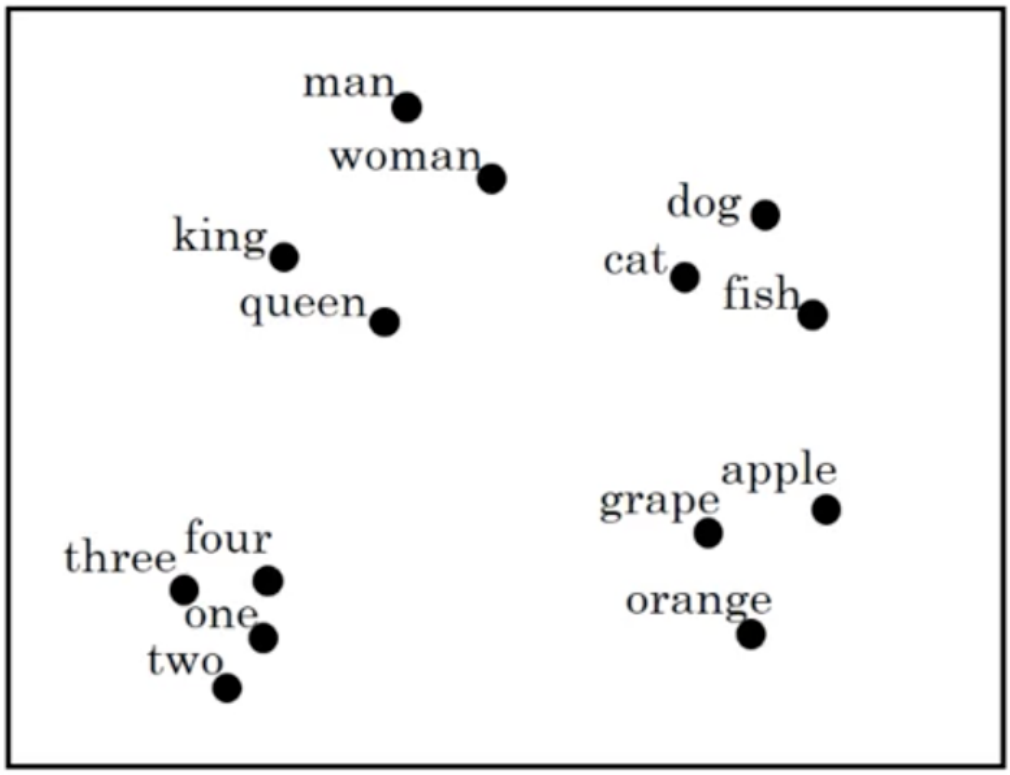
> Where f_n is the number of features.

Thinking back to our previous example, notice that our representations for the tokens "*apple*" and "*orange*" become quite similar. This is the critical point, and what allows our language model to generalize between word tokens and even entire sentences.

> In the later videos, we will see how to learn these embeddings. Note that the learned representations do not have an easy interpretation like the dummy embeddings we presented above.

Visualizing word embeddings

Once these feature vectors or *embeddings* are learned, a popular thing to do is to use dimensionality reduction to *embed* them into a 2D geometric space for easy visualization. An example of this using our word representations presented above:

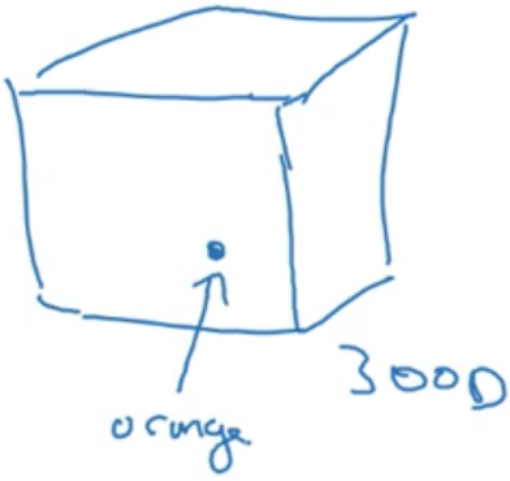


[https://postimg.cc/image/ofybhwcbz/]

We notice that semantically similar words tend to cluster together, and that each cluster seems to roughly represent some idea or concept (i.e., numbers typically cluster together). This demonstrates our ability to learn *similar* feature vectors for *similar* tokens and will allow our models to generalize between words and even sentences.

> A common algorithm for doing this is the t-SNE [http://www.wikiwand.com/en/T-distributed_stochastic_neighbor_embedding] algorithm.

The reason this feature representations are called *embeddings* is because we imagine that we are *embedding* each word into a geometric space (say, of 300 dimensions). If you imagine a cube, we can think of giving each word a single unit of space within this cube.



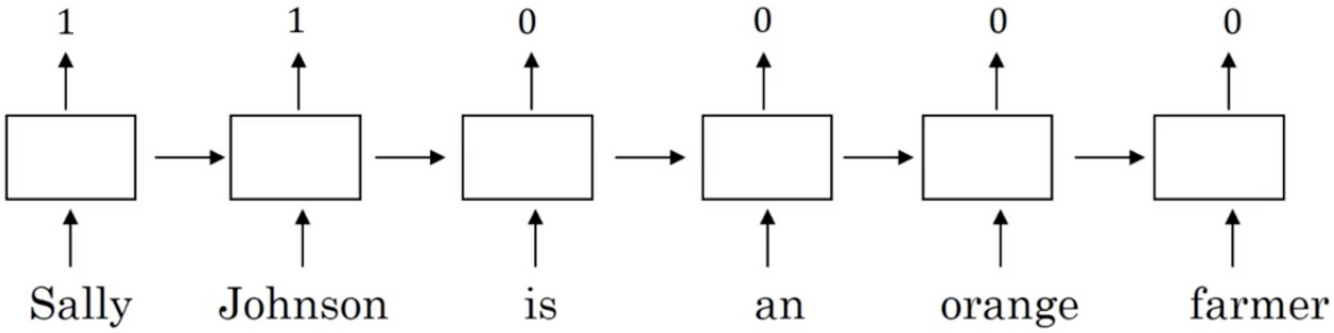
[https://postimg.cc/image/llv64kzwv/]

Introduction to word embeddings: Using word embeddings

In the last lecture, you saw what it might mean to learn a featurized representations of different words. In this lecture, you see how we can take these representations and plug them into NLP applications.

Named entity recognition example

Take again the example of named entity recognition, and image we have the following example:



[https://postimg.cc/image/9clc5o29r/]

Let's assume we correctly identify "*Sally Johnson*" as a PERSON entity. Now imagine we see the following sequence:

$$x : \text{"Robert Lin is a durian cultivator"}$$

> Note that durian is a type of fruit.

In all likelihood, a model using word embeddings as input should be able to generalize between the two input examples, a take advantage of the fact that it previously labeled the first two tokens of a similar training example ("*Sally Johnson*") as a PERSON entity. But how does the model generalize between "*orange farmer*" and "*durian cultivator*"?

Because word embeddings are typically trained on *massive* unlabeled text corpora, on the scale of 1 - 100 billion words. Thus, it is likely that the word embeddings would have seen and learned the similarity between word pairs ("*orange*", "*durian*") and ("*farmer*", "*cultivator*").

In truth, this method of **transfer learning** is typically how we use word embeddings in NLP tasks.

Transfer learning and word embeddings

How exactly do we utilize transfer learning of word embeddings for NLP tasks?

1. Learn word embeddings from large text corpus (1-100B words), OR, download pre-trained embeddings online.
2. Transfer the embedding to a new task with a (much) smaller training set (say, 100K words).
3. (Optional): Continue to fine-tune the word embeddings with new data. In practice, this is only advisable if your training dataset is quite large.

This method of transfer learning with word embeddings has found use in NER, text summarization, co-reference resolution, and parsing. However, it has been less useful for language modeling and machine translation (especially when a lot of data for these tasks is available).

One major advantage to using word embeddings to represent tokens is that it reduces the dimensionality of our inputs, compared to the one-hot encoding scheme. For example, a typical vocabulary may be 10,000 or more word types, while a typical word embedding may be around 300 dimensions.

Introduction to word embeddings: Properties of word embeddings

By now, you should have a sense of how word embeddings can help you build NLP applications. One of the most fascinating properties of word embeddings is that they can also help with **analogy reasoning**. And while reasoning by analogy may not be, by itself, the most important NLP application, it helps to convey a sense of what information these word embeddings are capturing.

Analogies

Let us return to our previous example:

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

[https://postimg.cc/image/wglsy3sof/]

Say we post the question: "*Man is to women as king is to **what**?*"

Many of us would agree that the answer to this question is "Queen" (in part because of humans remarkable ability to reason by analogy [https://plato.stanford.edu/entries/reasoning-analogy/]). But can we have a computer arrive at the same answer using embeddings?

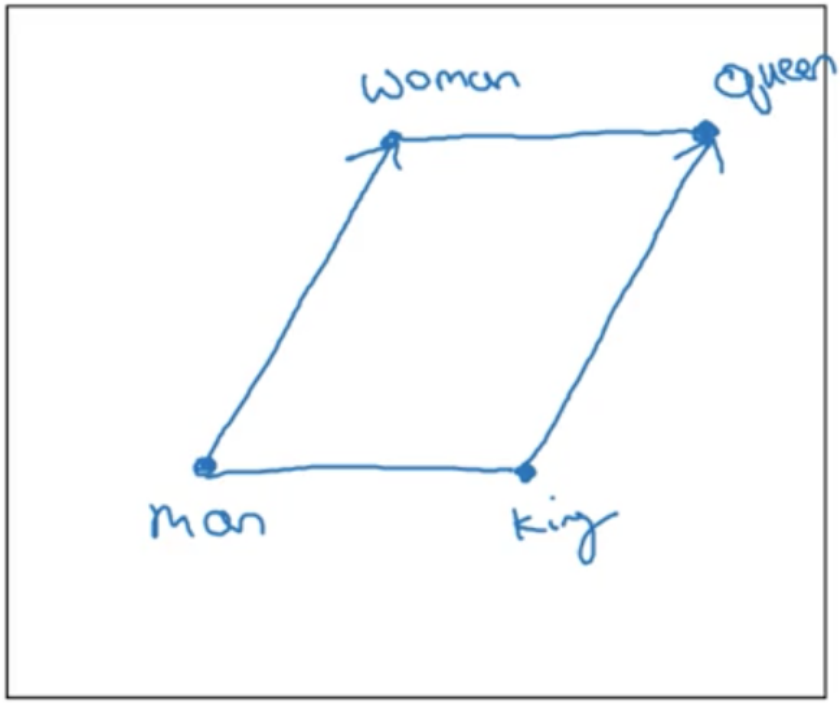
First, lets simplify our earlier notation and allow e_{man} to denote the learned embedding for the token "*man*". Now, if we take the difference $e_{man} - e_{woman}$, the resulting vector is closest to $e_{king} - e_{queen}$.

> Note you can confirm this using our made up embeddings in the table.

Explicitly, an algorithm to answer the question "*Man is to women as king is to **what**?*" would involve computing $e_{man} - e_{woman}$, and then finding the token w that produces $e_{man} - e_{woman} \approx e_{king} - e_{w}$.

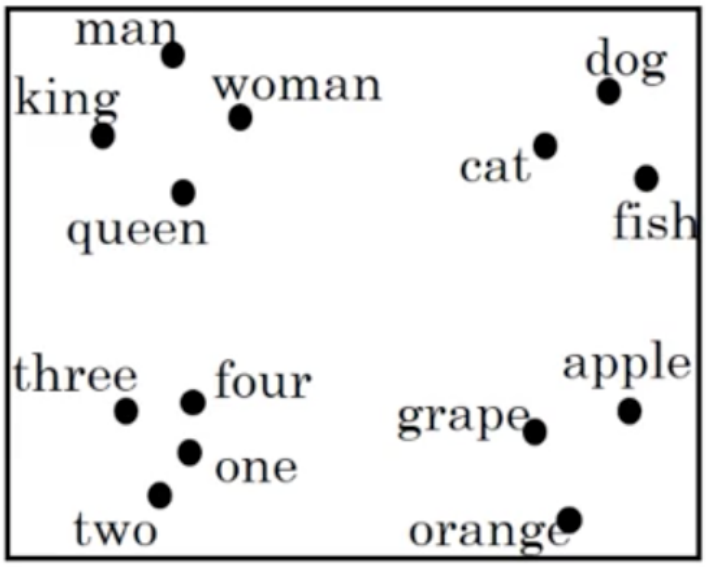
> This ability to mimic analogical reasoning and other interesting properties of word embeddings were introduced in this paper [https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/rvecs.pdf].

Lets try to visualize why this makes sense. Imagine our word embedding plotted as vectors in a 300D space (represented here in 2 dimensions for visualization). We would expect our vectors to line up in a parallelogram:



[https://postimg.cc/image/c1kq9ipfz/]

Note that in reality, if you use a dimensionality reduction algorithm such as t-SNE, you will find that this expected relationship between words in an analogy does not hold:



[https://postimg.cc/image/if9tcrmm7/]

We want to find $e_w \approx e_{king} - e_{man} + e_{woman}$. Our algorithm is thus:

$$argmax_w \ sim(e_w, e_{king} - e_{man} + e_{woman})$$

The most commonly used similarity function, sim is the *cosine similarity*:

$$sim(u, v) = \frac{u^T v}{||u||_2 ||v||_2}$$

Which represents the **cosine** of the angle between the two vectors u, v.

> Note that we can also use **euclidian distance**, although this is technically a measure of dissimilarity, so we need to take its negative. See here [http://blog.christianperone.com/2013/09/machine-learning-cosine-similarity-for-vector-space-models-part-iii/] for an intuitive explanation of the cosine similarity measure.

Introduction to word embeddings: Embedding matrix

Let's start to formalize the problem of learning a good word embedding. When we implement an algorithm to learn word embeddings, what we actually end up learning is an **embedding matrix**.

Say we are using a vocabulary V where $||V|| = 10,000$. We want to learn an embedding matrix E of shape $(300, 10000)$ (i.e., the dimension of our word embeddings by the number of words in our vocabulary).

$$E = \begin{bmatrix} e_{1,1} & \dots & e_{10000,1} \\ \dots & \dots & \\ e_{1,300} & & \dots \end{bmatrix}$$

> Where $e_{i,j}$ is the $j-th$ feature in the $i-th$ token.

Recall that we used the notation o_i to represent the one-hot encoded representation of the $i-th$ word in our vocabulary.

$$o_i = \begin{bmatrix} 0 \\ \dots \\ 1 \\ \dots \\ 0 \end{bmatrix}$$

If we take $E \cdot o_i$ then we are retrieving the embedding for the $i - th$ word in V, $e_i \in \mathbb{R}^{300 \times 1}$.

Summary

The import thing to remember is that our goal will be to learn an **embedding matrix** E. To do this, we initialize E randomly and learn all the parameters of this, say, 300 by 10,000 dimensional matrix. Finally, E multiplied by our one-hot vector o_i gives you the embedding vector for token i, e_i.

> Note that while this method of retrieving embeddings from the embedding matrix is intuitive, the matrix-vector multiplication is not efficient. In practice, we use a specialized function to lookup a column i of the matrix E, an embedding e_i.

Learning word embeddings: Learning word embeddings

Lets begin to explore some concrete algorithms for learning word embeddings. In the history of deep learning as applied to learning word embeddings, people actually started off with relatively *complex* algorithms. And then over time, researchers discovered they can use simpler and simpler algorithms and still get *very good* results, especially for a large dataset. Some of the algorithms that are most popular today are so simple that they almost seem little bit magical. For this reason, it's actually easier to develop our intuition by introducing some of the more complex algorithms first.

> Note that a lot of the ideas from this lecture came from Bengio et. al., 2003 [http://jmlr.org/papers/volume3/bengio03a/bengio03a.pdf].

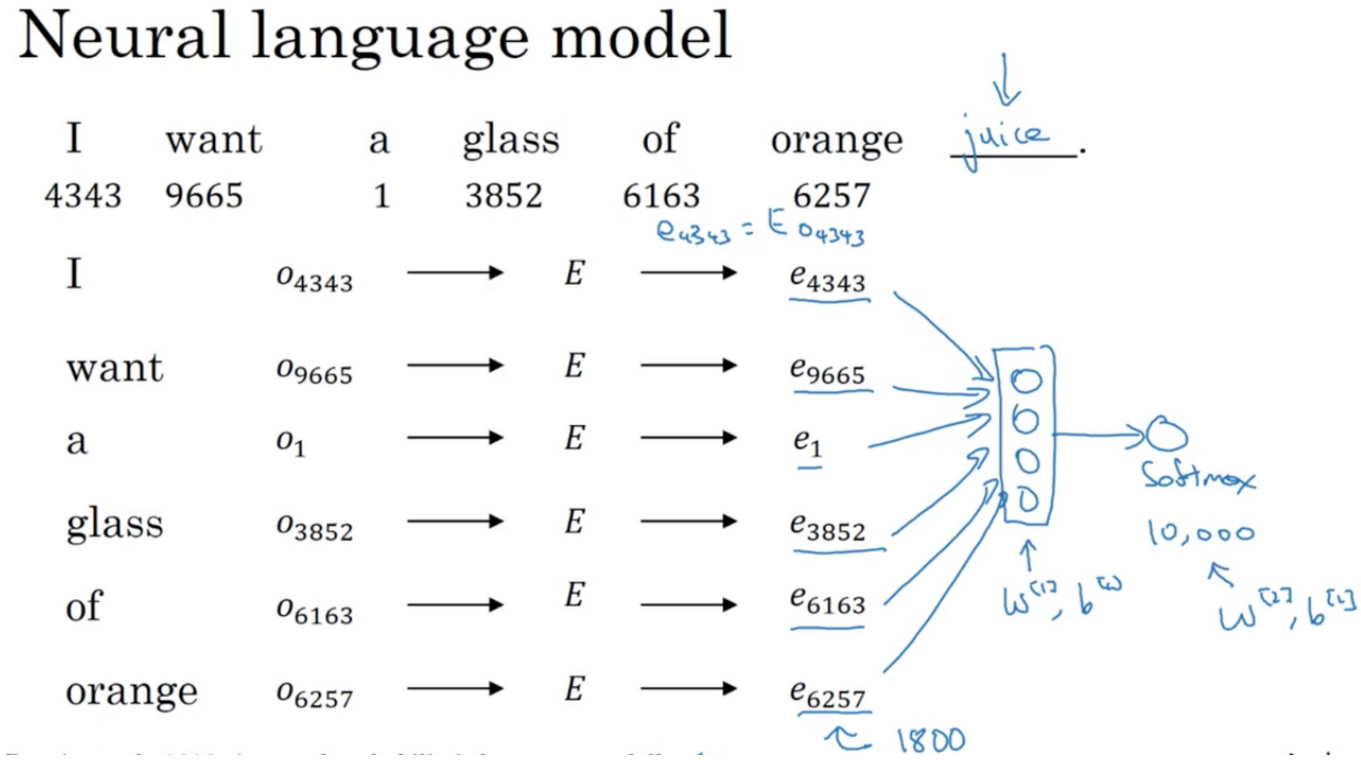
Algorithm 1

We will introduce an early algorithm for learning word embeddings, which was very successful, with an example. Lets say you are building a **neural language model**. We want to be able to predict the next word for any given sequence of words. For example:

> Note that, another common strategy is to pick a fixed window of words before the word we need to predict. The window size becomes a hyperparameter of the algorithm.

$$x : \text{"I want a glass of orange } \_\_\_\_\text{"}$$

One way to approach this problem is to lookup the embeddings for each word in the given sequence, and feed this to a densely connected layer which itself feeds to a single output unit with **softmax**.



[https://postimg.cc/image/78tghnh27/]

Imagine our embeddings are 300 dimensions. Then our input layer is $\mathbb{R}^{6\times300}$. Our dense layer and output softmax layer have their own parameters, $W^{[1]}, b^{[1]}$ and $W^{[2]}, b^{[2]}$. We can then use back-propagation to learn these parameters along with the embedding matrix. The reason this works is because the algorithm is incentivized to learn good word embeddings in order to generalize and perform well when predicting the next word in a sequence.

Generalizing

Imagine we wanted to learn the word "*juice*" in the following sentence:

\[x: \text{"I want a glass of orange *juice* to go along with my cereal"}\]

Typically, we would provide a neural language model with some *context* and have it predict this missing word from that context. There are many choices here:

- n words on the left & right of the word to predict
- last n word before the word to predict
- a single, *nearby* word

What researchers have noticed is that if your goal is to build a robust language model, choosing some n number of words before the target word as the context works best. However, if you goal is simply to learn word embeddings, then choosing other, simpler contexts (like a single, *nearby* word) work quite well.

To summarize, by posing the language modeling problem in which some **context** (such as the last four words) is used to predict some **target** word, we can effectively learn the input word embeddings via backprogopogation.

Learning Word Embeddings: Word2vec

In the last lecture, we used a neural language model in order to learn good word embeddings. Let's take a look at the the **Word2Vec** algorithm, which is a simpler and more computational efficient way to learn word embeddings.

> Most of the ideas in this lecture come from this paper: Mikolov et al., 2013 [https://arxiv.org/abs/1301.3781].

We are going to discuss the word2Vec **skip-gram** model for learning word embeddings.

Skip-gram

Let say we have the following example:

$$x : \text{"I want a glass of orange juice to go along with my cereal"}$$

In the skip-gram model, we choose (*context*, *target*) pairs in order to create the data needed for our supervised setting. To do this, for each *context* word we randomly choose a *target* word within some window (say, +/- 5 words).

Our learning problem:

$$x \rightarrow y$$

$$\text{Context }, c \text{ ("orange")} \rightarrow \text{Target}, t \text{ ("juice")}$$

The learning problem then, is to choose the correct *target* word within a window around the *context* word. Clearly, this is a very challenging task. However, remember that the goal is *not* to perform well on this prediction task, but to use the task along with backprogpogation to force the model to learn good word embeddings.

Model details

Lets take $||V|| = 10000$. Our neural network involves an embedding layer, E followed by a softmax layer, similar to the one we saw in the previous lecture:

$$E \cdot o_c \rightarrow e_c \rightarrow softmax \rightarrow \hat{y}$$

Our softmax layer computes:

$$p(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10000} e^{\theta_j^T e_c}}$$

> where θ_t is the parameters associated with output t and the bias term has been omitted.

Which is a $|V|$ dimensional vector containing the probability distribution of the target word being any word in the vocabulary for a given context word.

Our loss is the familiar negative log-likelihood:

$$\ell(\hat{y}, y) = -\sum_{i=1}^{10000} y_i \log \hat{y}_i$$

To *summarize*, our model looks up an embeddings in the embedding matrix which contains our word embeddings and is updated by backpropagation during learning. These embeddings are used by a softmax layer to predict a target word for a given context.

Problems with softmax classification

It turns out, there are a couple problems with the algorithm as we have described it, primarily due to the expensive computation of the *softmax* layer. Recall our softmax calculation:

$$p(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10000} e^{\theta_j^T e_c}}$$

Every time we wish to perform this softmax classification (that is, every step during training or testing), we need to perform a sum over $|V|$ elements. This quickly becomes a problem when our vocabulary reaches sizes in the milllions or even billions.

One solution is to use a **hierarchial softmax** classifier. The basic idea is to build a Huffman tree based on word frequencies. In this scheme, the number of computations to perform in the softmax layers scales as $\log |V|$ instead of V.

> I don't really understand this.

How to sample the context c?

Sampling our target words, t is straightforward once we have sampled their context, c, but how do we choose c itself?

Once solution is to sample uniform randomly. However, this leads to us choosing extremely common words (such as *the*, *a*, *of*, *and*, also known as stop words) much too often. This is a problem, as many updates would be made to e_c for these common words and many less updates would be made for less common words.

In practice, we use different heuristics to balance the sampling between very common and less common words.

Summary

In the original word2vec paper, you will find two versions of the word2vec model: the **skip-gram** one introduced here and another called **CBow**, the continuous bag-of-words model. This model takes the surrounding contexts from a middle word, and uses them to try to predict the middle word. Each model has its advantages and disadvantages.

The key problem with the **skip-gram** model as presented so far is that the softmax step is *very expensive* to calculate because it sums over the entire vocabulary size.

Learning Word Embeddings: Negative Sampling

In the last lecture, we saw how the **skip-gram** model allows you to construct a supervised learning task by mapping from contexts to targets, and how this in turn allows us to learn a useful word embeddings. The major the downside of this approach was that was the **softmax** objective was *very slow to compute*.

Lets take a look at a modified learning problem called **negative sampling**, which allows us to do something similar to the skip-gram model but with a much more efficient learning algorithm.

> Again, most of the ideas in this lecture come from this paper: Mikolov et al., 2013 [https://arxiv.org/abs/1301.3781].

Similar to the skip-gram model, we are going to create a supervised learning setting from unlabeled data. Explicitly, the problem is to predict whether or not a given pair of words is a *context*, *target* pair.

First, we need to generate training examples:

- **Positive** examples are generated exactly how we saw with the skip-gram model, i.e., by sampling a context word and choosing a target word within some window around the context.
- To generate the **negative examples**, we take a sampled context word and then for some k number of times, we choose a target word *randomly* from our vocabulary (under the assumption that this random word *won't* be associated with our sampled *context* word).

As an example, take the following sentence:

$$x : \text{"I want a glass of orange juice to go along with my cereal"}$$

Then we might construct the following (context, target) training examples:

- $(orange, juice, 1)$
- $(orange, king, 0)$

- $(orange, book, 0)$
- $(orange, the, 0)$
- $(orange, of, 0)$

Where 1 denotes a **positive** example and 0 a **negative** example.

> Note that this leads to an obvious problem: some of our randomly chosen target words in our generated negative examples will in fact be within the context of the sampled context word. It turns out this is OK, as much more often than not our generated negative examples are truly negative examples.

Next, we define a supervised learning problem, where our inputs x are these generated positive and negative examples, and our targets y are whether or not the input represents a true (*context*, *target*) pair (1) or not (0):

$$x \rightarrow y$$

$$(context, target) \rightarrow 1 \text{ or } 0$$

> Explicitly, we are asking the model to predict whether or not the two words came from a distribution generated by sampling from within a context (defined3 as some window around the words) or a distribution generated by sampling words from the vocabulary at random.

How do you choose k? Mikolov *et. al* suggest $k = 5 - 20$ for small datasets, and $k = 2 - 5$ for large datasets. You can think of k as a 1:k ratio of **positive** to **negative** examples.

Model details

Recall the softmax classifier from the skip-gram model:

$$\text{Softmax: } p(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10000} e^{\theta_j^T e_c}}$$

For our model which uses negative sampling, first define each input, output pair as c, t and y respectively. Then, we define a logistic regression classifier:

$$p(y = 1|c, t) = \sigma(\theta_t^T e_c)$$

Where θ_t represents the parameter vector for a possible target word t, and e_c the embedding for each possible context word.

> NOTE: totally lost around the 7 min mark. Review this.

This technique is called **negative sampling** because we generate our training data for the supervised learning setting by first creating a positive example and then *sampling k negative* examples.

Selecting negative examples

The final import point is how we *actually* sample **negative** examples in *practice*.

- One option is to sample the target word based on the empirical frequency of words in your training corpus. The problem of this solution is that we end up sampling many highly frequent stop words, such as "and", "of", "or", "but", etc.
- Another extreme is to sample the negative examples uniformly random. However, this also leads to a very non-representive sampling of target words.

What the authors [https://arxiv.org/abs/1301.3781] found to work best is something in between:

$$P(w_i) = \frac{f(w_i)^{\frac{3}{4}}}{\sum_{j=1}^{|V|} f(w_j)^{\frac{3}{4}}}$$

Here, we sample proportional to the frequency of a word to the power of $\frac{3}{4}$. This is somewhere between the two extremes of sampling words by their frequency and sampling words at uniform random.

Summary

To summarize,

- we've seen how to learn word vectors with a **softmax classier**, but it's very computationally expensive.
- we've seen that by changing from a softmax classification to a bunch of binary classification problems, we can very efficiently learn words vectors.
- as is the case in other areas of deep learning, there are open source implementations of the discussed algorithms you can use to learn these embeddings. There are also pre-trained word vectors that others have trained and released online under permissive licenses.

Learning Word Embeddings: GloVe word vectors

The final algorithm we will look at for learning word embeddings is **global vectors for word representation** (**GloVe**). While not used as much as **word2vec** models, it has its enthusiasts -- in part because of its simplicity.

> This algorithm was original presented here [http://www.aclweb.org/anthology/D14-1162].

Previously, we were sampling pairs of words (*context*, *target*) by picking two words that appear in close proximity to one another in our text corpus. In the GloVe algorithm, we define:

- X_{ij}: the number of times word i appears in the context of word j.
- X_{ij} == X_{ji}

> Note that X_{ij} == X_{ji} is not necessarily true in other algorithms (e.g., if we were to define the context as being the immediate next word). Notice that i and j play the role of c and t.

Model

The models objective is as follows:

Minimize $\sum_{i=1}^{|V|} \sum_{j=1}^{|V|} f(X_{ij})(\theta_i^T e_j + b_i + b'_j - \log X_{ij})^2$

- Think of $\theta_i^T e_j$ as a measure of how similar two words are, based on how often the occur together: $\log X_{ij}$. More specifically, we are trying to minimize this difference using gradient descent by searching for the pair of words i, j whose inner product $\theta_i^T e_j$ is a good predictor of how often they are going to appear together, $\log X_{ij}$.
- If $X_{ij} = 0$, $\log X_{ij} = \log 0 = -\infty$ which is undefined. We use $f(X_{ij})$ as weighting term, which is 0 when X_{ij} = 0, so we don't sum over pairs of words i, j when $X_{ij} = 0$. $f(X_{ij})$ is also used to weight words, such that extremely common words don't "drown" out uncommon words. There are various heuristics for choosing $f(X_{ij})$. You can look at the original paper [http://www.aclweb.org/anthology/D14-1162] for details for how to choose this heuristic.

> Note, we use the convention $0 \log 0 = 0$

Something to note about this algorithm is that the roles of $theta$ and e are now completely *symmetric*. So, θ_i and e_j 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. In fact, one way to train the algorithm is to initialize θ

and e both uniformly and use gradient descent to minimize its objective, and then when you're done for every word, to then take the average:

$$e_w^{final} = \frac{e_w + \theta_w}{2}$$

because $theta$ and e in this particular formulation play symmetric roles unlike the earlier models we saw in the previous videos, where $theta$ and e actually play different roles and couldn't just be averaged like that.

A note of the featurization view of word embeddings

Recall that when we first introduced the idea of word embeddings, we used a sort of *featurization view* to motivate the reason why we learn word embeddings in the first place. We said, "Well, maybe the first dimension of the vector captures gender, the second, age...", so forth and so on.

However in practice, we cannot guarantee that the individual components of the embeddings are interpretable. Why? Lets say that there is some "space" where the first axis of the embedding vector is gender, and the second age. There is no way to guarantee that the actual dimension for each "feature" that the algorithm arrives at will be easily interpretable by humans. Indeed, if we consider the learned representation of each context, target pair, we note that:

$$\theta_i^T e_j = (A\theta_i)^T (A^{-T} e_j) = \theta_i^T A^T A^{-T} e_j$$

Where A is some arbitrary invertible matrix. The key take away is that the dimensions learned by the algorithm are not human interpretable, and each dimension typically encodes *some part* of what we might think of a feature, as opposed to encoding an entire feature itself.

Applications using Word Embeddings: Sentiment Classification

Sentiment classification is the task of looking at a piece of text and telling if someone likes or dislikes the thing they're talking about. It is one of the most important building blocks in NLP and is used in many applications. One of the challenges of sentiment classification is a lack of labeled data. However, with word embeddings, you're able to build good sentiment classifiers even with only modest-size label training sets. Lets look at an example:

$$x : \text{"The dessert is excellent."}, y : 4/5 \text{ stars}$$

$$x : \text{"Service was quite slow."}, y : 2/5 \text{ stars}$$

$$x : \text{"Good for a quick meal, but nothing special."}, y : 3/5 \text{ stars}$$

$$x : \text{"Completely lacking in good taste, good service, and good ambience."}, y : 1/5 \text{ stars}$$

> While we are using restaurant reviews as an example, sentiment analysis is often applied to voice of the customer [http://www.wikiwand.com/en/Voice_of_the_customer] materials such as reviews and social media.

Common training set sizes for sentiment classification would be around 10,000 to 100,000 words. Given these small training set sizes, word embeddings can be extremely useful. Lets use the above examples to introduce a couple of different algorithms

Simple sentiment classification model

Take,

$$x : \text{"The dessert is excellent."}, y : 4/5 \text{ stars}$$

As usual, we map the tokens in the input examples to one-hot vectors, multiply this by a pre-trained embedding matrix and obtain our embeddings, e_w. Using a pre-trained matrix is essentially a form of transfer learning, as we are able to encode information learned on a much larger corpus (say, 100B tokens) and use it for learning on a much smaller corpus (say, 10,000 tokens).

We could then *average* or *sum* these embeddings, and pass the result to a *softmax* classifier which outputs $\hat{y}$, the probability of the review being rated as 1, 2, 3, 4 or 5 stars.

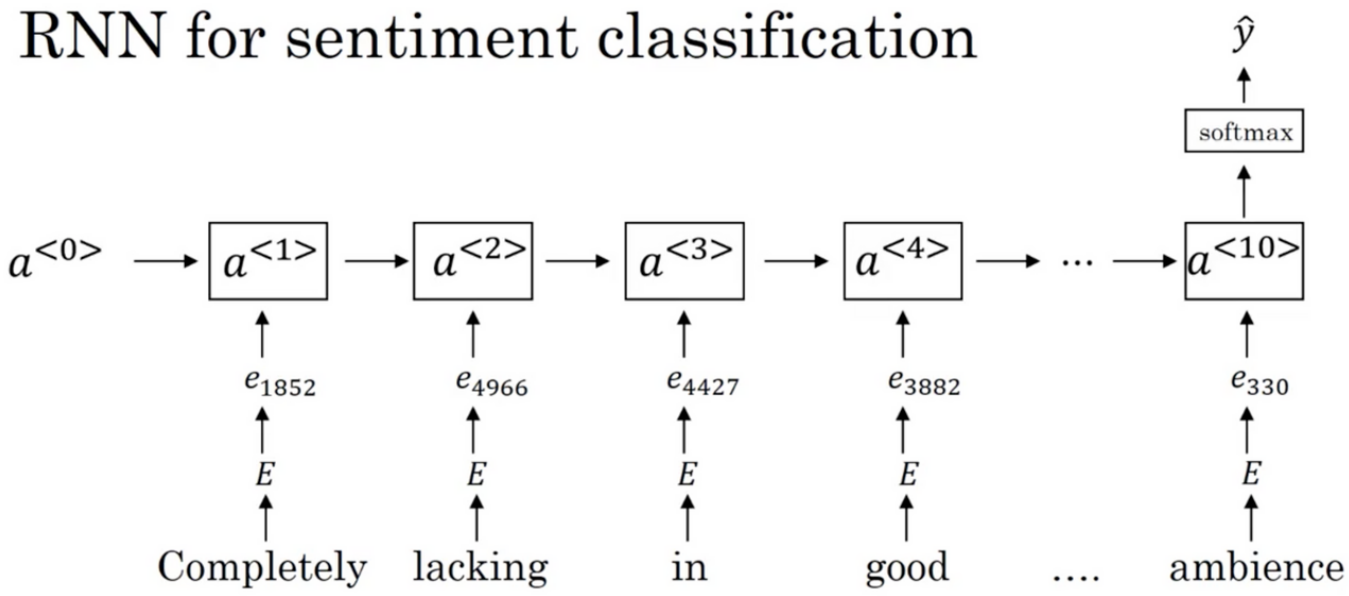
This algorithm will work OK, but fails to capture *negation* of positive words (as it does not take into account word order). For example:

$$x : \text{"Completely lacking in good taste, good service, and good ambience."}, y : 1/5 stars$$

might incorrectly be predicted to correspond with a high star rating, because of the appearance of "good" three times.

RNN sentiment classification model

A more sophisticated model involves using the embeddings as inputs to an RNN, which uses a softmax layer at the last timestep to predict a star rating:



[https://postimg.cc/image/5zlev64mn/]

Recall that we actually saw this example when discussing many-to-one RNN architectures. Unlike the previous, simpler model, this model takes into account word order and performs much better on examples such as:

$$x : \text{"Completely lacking in good taste, good service, and good ambience."}, y : 1/5 \text{ stars}$$

which contain many negated, positive words. When paired with pre-trained word embeddings, this model works quite while.

Summary

Pre-trained word embeddings are especially useful for NLP tasks where we don't have a lot of training data. In this lecture, we motivated that idea by showing how pre-trained word embeddings can be used as inputs to very simple models to perform sentiment classification.

Applications using Word Embeddings: Debiasing word embeddings

Machine learning and AI algorithms are increasingly trusted to help with, or to make, extremely important decisions. As such, we would like to make sure that, as much as possible, they're free of undesirable forms of bias, such as gender bias, ethnicity bias and so on. Lets take a look at reducing bias in word embeddings.

> Most of the ideas in this lecture came from this paper [http://papers.nips.cc/paper/6228-man-is-to-computer-programmer-as-woman-is-to-homemaker-debiasing-word-embeddings.pdf].

When we first introduced the idea of word embeddings, we leaned heavily on the idea of analogical reasoning to build our intuition. For example, we were able to ask "Man is to woman as king is to ____?" and using word emebddings, arrive at the example queen. However, we can also ask other questions that reveal a *bias* in embeddings. Take the following analogies encoding in some learned word embeddings:

$$\text{"Man is to computer programmer as woman is to homemaker"}$$

$$\text{"Father is to doctor as mother is to nurse"}$$

Clearly, these embeddings are encoding unfortunate gender stereotypes. Note that these are only examples, biases against ethnicity, age, sexual orientation, etc. can also become encoded by the learned word embeddings. In order for these biases to be learned by the model, they must first exist in the data used to train it.

Addressing bias in word embeddings

Lets say we have already learned 300D embeddings. We are going to stick to gender bias for simplicities sake. The process for debiasing these embeddings is as follows:

1 **Identify bias direction**:

Take a few examples where the only difference (or only major difference) between word embeddings is gender, and subtract them:

- $e_{he} - e_{she}$
- $e_{male} - e_{female}$
- ...

Average the differences. The resulting vector encodes a 1D subspace that may be the **bias** axis. The remaining 299 axes are the **non-bias direction**

> Note in the original paper, averaging is replaced by SVD, and the **bias** axis is not necessarily 1D.

2 **Neutralize**:

For every word that is not definitional, project them onto **non-bias direction** or axis to get rid of bias. These do **not** include words that have a legitimate gender component, such as *"grandmother"* but **do** include words for which we want to eliminate a learned bias, such as *"doctor"* or *"babysitter"* (in this case a gender bias, but it could also be a sexual orientation bias, for example).

Choosing which words to neutralize is challenging. For example, *"beard"* is characteristically male, so its likely not a good idea to neutralize it with respect to gender. The authors of the original paper actually trained a classifier to determine which words were definitional with respect to the bias (in our case gender). It turns out that english does not contain many words that are definitional with respect to gender.

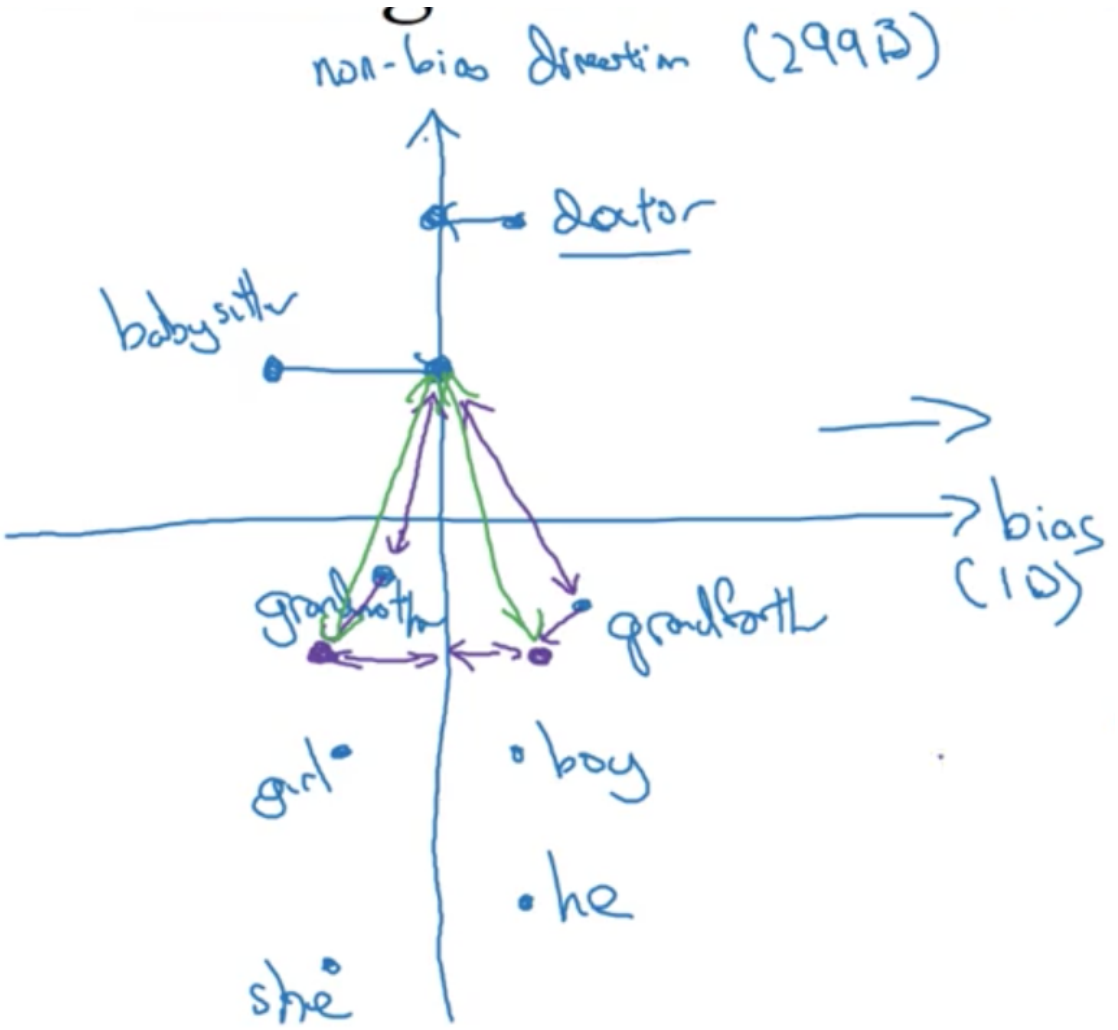
3 **Equalize pairs**:

Take pairs of definitional words (such as *"grandmother"* and *"grandfather"* and equalize their difference to the **non-bias direction** or axis. This ensures that these words are equidistant to all other words for which we have "neturalized" and

encoded bias.

This process is a little complicated, but the end results is that these pairs of words, (e.g. *"grandmother"* and *"grandfather"*) are moved to a pair of points that are equidistant from the **non-bias direction** or axis.

It turns out, the number of these pairs is very small. It is quite feasible to pick this out by hand.



[https://postimg.cc/image/g3h2n9z33/]

Summary

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

Week 3: Sequence models & Attention mechanism

In this series, we will look primarily at sequence models, which are useful for everything from machine translation to speech recognition. We will also look at attention models.

Sequence models & Attention mechanism: Basic Models

Machine translation

Lets say we want to translate from *french* to *english*. We will use the following examples:

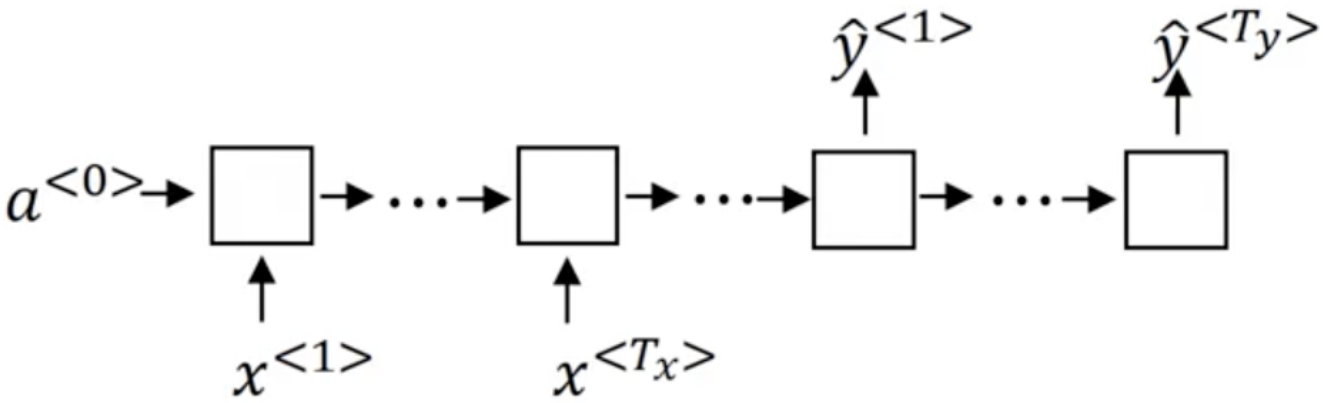
$$x : \text{"Jane visite l'Afrique en septembre"}$$

$$y : \text{"Jane is visiting Africa in September"}$$

Recall that we represent individual elements in the input sequence as $x^{<t>}$, and in the output sequence as $y^{<t>}$.

> Most of the ideas presented in this lecture are from this paper [http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks] and this paper [https://arxiv.org/abs/1406.1078].

First, we build our **encoder**, an RNN (LSTM or GRU) which processes the *input* sequence. The encoder outputs a vector that represents the learned representation of the input sequence. A **decoder** takes this *output* sequence, and outputs the translation one word at a time.



> Note that the outputs from each timestep in the decoder network are actually passed as input for the next timestep in the case of language modelling.

Given enough data, this **sequence to sequence** architecture actually works quite well.

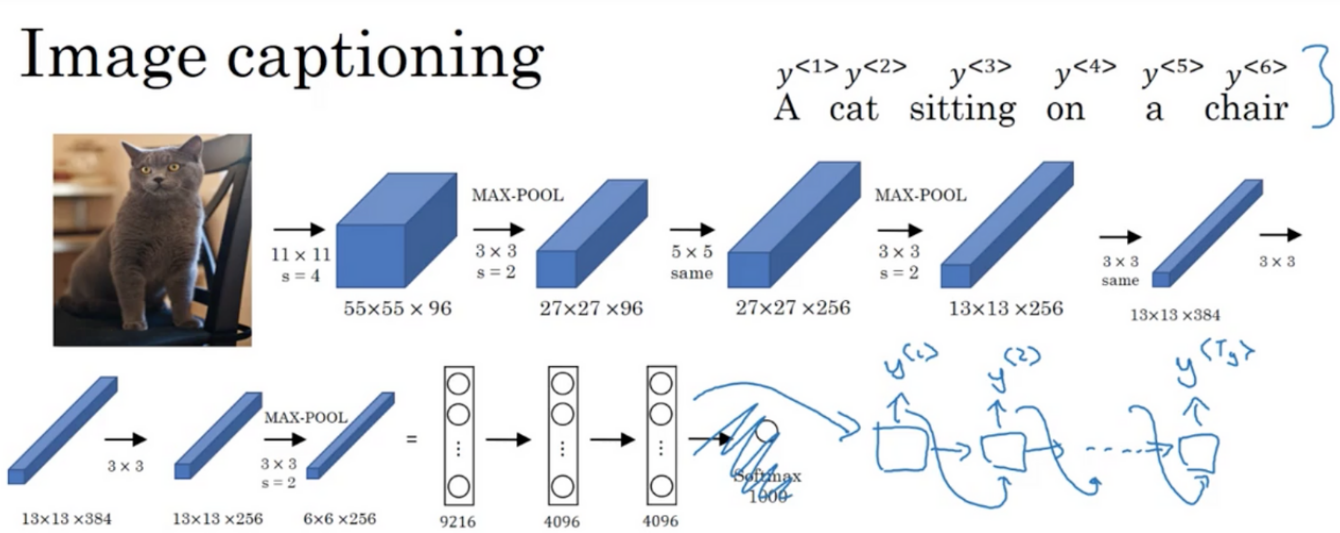
Image captioning

The task of **image captioning** involves inputing an image, and having the network generate a natural language caption.

$$x : \text{an image of a cat of a chair}$$

$$y : \text{"A cat sitting on a chair"}$$

Typically, we use a CNN as our **encoder**, without any classification layer at the end. We feed the learned representation to a **decoder**, an RNN, which generates and output sequence which is our image caption.



[https://postimg.cc/image/w49o0izm7/]

Summary

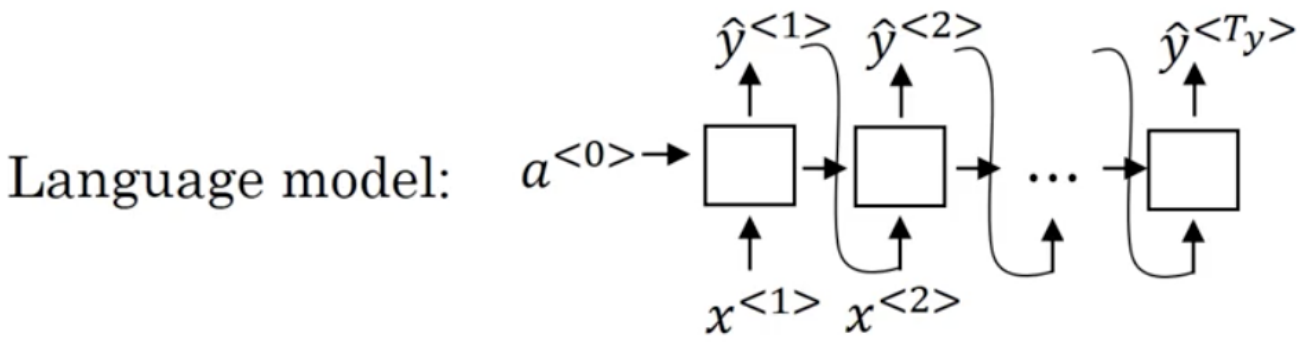
- Simple sequence to sequence (seq2seq) models are comprised of an **encoder** and **decoder**, which themselves are neural networks (typically recurrent or convolutional).
- Seq2seq architectures are able to preform reasonably well when given enough data. With certain modifications (attention) and additions (beam search), they are able to achieve state-of-the-art performance on tasks like **machine translation** and **image captioning**.
- These architecture difference slightly to some of the sequence generation architectures we saw previously however, and we will explore the difference in the next lecture.

Sequence models & Attention mechanism: Picking the most likely sentence

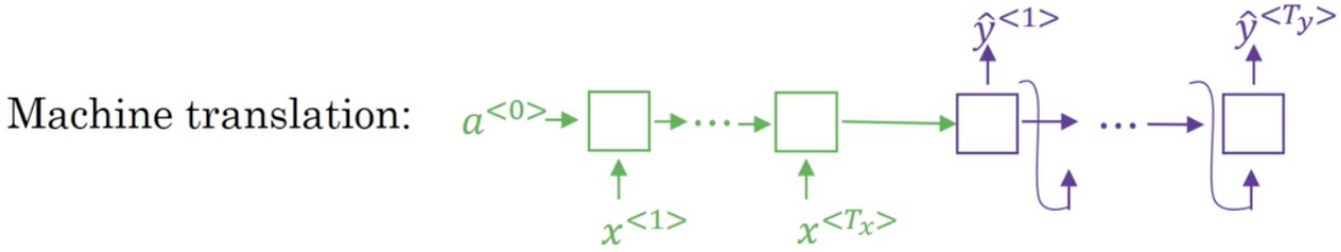
While there are some similarities between the **sequence to sequence machine translation model** and the **language models** that you have worked within the first week of this course, there are some significant differences as well.

Machine translation as a conditional language model

We can think of machine translation as building a *conditional language model*. First, recall the language model we worked with previously:



The machine translation model looks like:



Notice that the decoder network looks pretty much identical to the language model. However, instead of the initial hidden state being $a^{<t>} = 0$, the initial hidden state is initialized with the learned representation from the encoder. That is why we can think of this as a *conditional language model*. Instead of modeling the probability of any sentence, it is now modeling the probability of say, the output English translation, conditioned on some input French sentence.

Finding the most likely translation

So how does this work in practice? Our model produces the following:

$$P(y^{<1>}, \dots, y^{<T_y>}|x)$$

The probability of different corresponding translated sentences based on the input sentence, x. Unlike the language model we saw earlier, we *do not want to sample words randomly*. Instead, we want to choose the sentence y that maximizes $P(y^{<1>}, \dots, y^{<T_y>}|x)$,

$$argmax_y P(y^{<1>}, \dots, y^{<T_y>}|x)$$

The most common algorithm for solving this problem is called the **beam search**, which we will look at in the next lecture. Before we look at beam search, you might be wondering, why not use a **greedy search**? In our case, this would involve choosing the most likely output, $\hat{y}^{<t>}$ at each timestep, one timestep at a time, starting from $t = 1$.

In simple terms, this approach does *not* maximize the **joint probability** $P(y^{<1>}, \dots, y^{<T_y>}|x)$, which is what we are really after. To see why this so, take our input example:

$$x : \text{"Jane visite l'Afrique en septembre"}$$

And our outputs generated by **beam search** and **greedy search** respectively:

$$beam : \text{"Jane is visiting Africa in September"}$$

$$greedy : \text{"Jane is going to be visiting Africa in September"}$$

In greedy search, we are optimizing the output at each timestep without regard for the overall sequence. As such, we will often produce less succicent, and more verbose sentences. In this case, $p(\text{"Jane is"}|\text{"going"}) > p(\text{"Jane is"}|\text{"visiting"})$, which is why we produce the more verbose "*is going to be visiting*" as opposed to "*is visiting*".

> While this is a contrived example, it captures a larger phenomena. When generating sequences, it is often a good idea to maximize the joint probability across the entire sequence. Otherwise, we end up with sub-optimal performance tasks in which there is strong dependencies between elements of a sequence.

Another important point, is that the space of all possible output sentences is huge. For a ten-word sentence drawn and a vocabulary of 10,000 words, we have $10,000^{10}$ possible sentences. For this reason, we need an **approximate search algorithm** (a heuristic [http://www.wikiwand.com/en/Heuristic_(computer_science)]). While this won't always succeed in finding the sentence $\hat{y}$ that maximizes that conditional probability, it will typically do a "good enough" job without needed to enumerating all possible sentences.

Summary

- In this lecture, you saw how machine translation can be posed as a *conditional language modeling problem*.
- One major difference between this and the earlier language modeling problems is rather than generating a sentence at random, you try to find the most likely translation.
- The set of all sentences of a certain length is too large to exhaustively enumerate. So, we have to resort to an approximate search algorithm.

Sequence models & Attention mechanism: Beam Search

Recall that, for a machine translation system given an input French sentence, you don't want to output a random English translation, you want to output the best and the most likely English translation. The same is also true for **speech recognition** where, given an input audio clip, you don't want to output a *random* text transcript of that audio, you want to output the *best*, maybe the *most likely*, text transcript. Beam search is the most widely used algorithm to do this.

Beam search algorithm

Step 1

The first thing beam search needs to do is pick the first word, $\hat{y}^{<1>}$ of the translation given our input sequence x, $P(\hat{y}^{<1>}|x)$. Contrary to greedy search however, we can evaluate multiple choices at the same time. The number of choices is designated by a parameter of the algorithm, the **beam width** B. Lets say for our purposes that we choose $B = 3$.

In practice, we run our sentence to be translated through the encoder-decoder network and store in memory the three most likely, first words of the translated sentence.

Step 2

For each of these three choices, we consider what should be the second word in the translated output sequence, $\hat{y}^{<2>}$. Recall that the previously selected output, $\hat{y}^{<1>}$ is fed as input to the next timestep in the decoder network.

In practice, we are looking to compute:

$$P(\hat{y}^{<1>},\ \hat{y}^{<2>}|x) = P(\hat{y}^{<1>}|x)P(\hat{y}^{<2>}|x,\ \hat{y}^{<1>})$$

Where $P(\hat{y}^{<1>}|x)$ is computed by the decoder network in the first step and $P(\hat{y}^{<2>}|x,\ \hat{y}^{<1>})$ computed by the decoder network in the second step. In total, we evaluate $B \cdot |V|$ possible choices for the second word, but we only save the top B most likely choices for $\hat{y}^{<1>},\ \hat{y}^{<2>}$. We may actually end up dropping one or more possible choices we previously made for $\hat{y}^{<1>}$.

An implementation detail: we instantiate B copies of the seq2seq model to compute $P(\hat{y}^{<1>}|x)P(\hat{y}^{<2>}|x,\ \hat{y}^{<1>})$, one for each choice of $\hat{y}^{<1>}$.

Step n

We continue this process, at each step n computing $P(\hat{y}^{<1>}, \hat{y}^{<2>}, \dots \hat{y}^{<n>}|x)$ using the chain rule for conditional probabilities:

$$P(\hat{y}^{<1>}, \hat{y}^{<2>}, \dots \hat{y}^{<n>}|x) = P(\hat{y}^{<n>}|\hat{y}^{<1>}, \dots, \hat{y}^{<n-1>}, x) \dots P(\hat{y}^{<1>}|x)$$

We store the top B most likely sequences $\hat{y}^{<1>}, \hat{y}^{<2>}, \dots \hat{y}^{<n-1>}$ and their corresponsding seq2seq model to use in the next step, where we again evaluate $|V|$ possible words for our B number of previous choices.

The outcome of this process, hopefully, is a prediction which is terminated by the special token. Notice that, when $B = 1$, this process essentially becomes the greedy search algorithm seen previously.

Sequence models & Attention mechanism: Refinements to Beam Search

In the last lecture, we saw the basic beam search algorithm. In this lecture, we will look at some little changes that make it work even better.

Length normalization

Recall that in beam search, we are trying to compute the following:

$$argmax_y \prod_{t=1}^{T_y} P(y^{<t>}|x, y^{<1>}, \dots, y^{<t-1>})$$

The product term is capturing the idea that $P(y^{<1>}, \dots, y^{<T_y>}|x) = P(y^{<1>}|x) \dots P(y^{<T_y>}|x, y^{<1>}, \dots, y^{<T_y-1>})$, which itself is just the chain rule for conditional probabilities.

The problem here, is that many many small probabilities can lead to numerical underflow [http://www.wikiwand.com/en/Arithmetic_underflow]. For this reason, we take the log of the product (which becomes the sum of the log of the individual elements):

$$argmax_y \sum_{t=1}^{T_y} \log P(y^{<t>}|x, y^{<1>}, \dots, y^{<t-1}>)$$

This leads to a more numerically stable algorithm, which reduces the chance of rounding errors.

> Because $\log P(y|x)$ and $P(y|x)$ are monotonically increasing functions, the value that maximizes one also maximizes the other.

Notice as well that the optimization objective actually favors short sentences (this is true of both formulations listed above). In machine translation, for example, This has the negative consequence of promoting short translated sentences over long translated sentences even when the longer sentence is preferable. A solution to this problem is to *normalize* our objective objective with respect to output length:

$$argmax_y \frac{1}{T_y^\alpha} \sum_{t=1}^{T_y} \log P(y^{<t>}|x, y^{<1>}, \dots, y^{<t-1}>)$$

In practice, we typically introduce a new parameter α which *softens* the length normalization, to retain a penalty on very very long output sequences.

Beam search recap

So, in full:

As you run beam search you see a lot of sentences with length equal 1, 2, 3, and so on. Maybe you run beam search for 30 steps and you consider output sentences up to length 30, let's say. With a beam width of 3, you will be keeping track of the top three possibilities for each of these possible sentence lengths, 1, 2, 3, 4 and so on, up to 30. Then, you would look at all of the output sentences and score them against our objective:

$$argmax_y \frac{1}{T_y^\alpha} \sum_{t=1}^{T_y} \log P(y^{<t>}|x, y^{<1>}, \ldots, y^{<t-1>})$$

Finally, of all of these sentences that we encounter through beam search, we pick the one that achieves the highest value on this normalized log probability objective (sometimes called a **normalized log likelihood objective**). This brings us to the final translation, your outputs.

Beam width discussion

Finally, an implementational detail. *How do you choose the beam width* B? The larger B is, the more possibilities you're considering, and thus the better the sentence you will probably find. But the larger B is, the more computationally expensive your algorithm is, because you're also keeping a lot more possibilities around.

So here are the pros and cons of setting B to be very large versus very small:

- If the beam width is very large, then you consider a lot of possibilities, and so you tend to get a better result, *but* it will be slower, the memory requirements will also grow, and it will also be computationally slower.
- If you use a small beam width, then you are likely to get a worse result because you're keeping less possibilities in mind as the algorithm is running. But you get a result faster and the memory requirements will be lower.

In the previous video, we used in our running example a beam width of three. In practice, that is on the small side. In production systems, it's not uncommon to see a beam width maybe around 10 -- 100 would be considered very large for a production system. For research systems, where people want to squeeze out every last drop of performance in order to publish a paper, tt's not uncommon to see people use beam widths of 1,000 to 3,000. In truth, you just need to try a variety of values of B as you work through your application. Beware that as B gets very large, there is often diminishing returns.

> Expect to see a huge gain as you go from a beam width of 1, which is essentially greedy search, to 3, to maybe 10, but gains may diminish from there on out.

Sequence models & Attention mechanism: Error analysis on beam search

In the third course of this sequence of five courses, we saw how **error analysis** can help you focus your time on doing the most useful work for your project. Because **beam search** is an approximate search algorithm, also called a *heuristic* [http://www.wikiwand.com/en/Heuristic_(computer_science)], it doesn't always output the most likely sentence. *So what if beam search makes a mistake?* In this lecture, you'll learn how error analysis interacts with beam search and how you can figure out whether it is the beam search algorithm that's causing problems and worth spending time on, *or* whether it might be your RNN model that is causing problems and worth spending time on.

Lets use the following, simple example to see how error analysis works with beam search:

$$x : \text{"Jane visite l'Afrique en septembre"}$$

Assume further that our human-provided ($y*$) and machine provided ($\hat{y}$) translations for this sentence are:

$$y* : \text{"Jane is visiting Africa in September"}$$

$$\hat{y} : \text{"Jane visited Africa last September"}$$

Our model has to two main components: our encoder-decoder architecture (which we will simply refer to as our 'RNN' from here on out) and the beam search algorithm applied to the outputs of this RNN. It would be extremely helpful if we could assign blame to *one of these components individually* when we get a bad translation, like our example $\hat{y}$.

Similar to how it may be tempting to collect more training data when our models underperform (it 'never hurts'!) it is also tempting in this case to increase the beam width (again, it never seems to hurt!). Error analysis is a more principled way to improve our model, by helping us focus our attention on what is currently hurting performance the most.

Error analysis applied to machine translation

Recall that the RNN computes $P(y|x)$. The first step in our analysis is to compute $P(y*|x)$ and $P(\hat{y}|x)$. We then ask, which probability is larger? If

Case 1: $P(y*|x)$ > $P(\hat{y}|x)$

Beam search chose $\hat{y}$ (recall, its optimization objective is $\hat{y} = argmax_y P(y|x)$), but $y*$ is more likely according to your model.

Conclusion: We conclude that beam search is at fault. It faild to find a value for y which maximizes $P(y|x)$.

Case 2: $P(y*|x) \leq P(\hat{y}|x)$

$y*$ is a better translation than $\hat{y}$. But our RNN predicted $P(y*|x) \leq P(\hat{y}|x)$

Conclusion: We conclude that RNN model is at fault. It failed to model a better translation as being more likely than a worse translation for a given input sentence.

> Note, we are ignoring length normalization here for simplicity. In reality, you would use the entire optimization objective with length normalization instead of just $P(y|x)$ in your error analysis.

In practice, we might collect some examples of our gold translations (provided by a human) and compare them in a table to the corresponding machine-provided translations, tallying $P(y*|x)$ and $P(\hat{y}|x)$ for each example. We can then use the rules provided above to assign blame to either the RNN or to beam search in each case:

Human	Algorithm	$P(y^*\|x)$	$P(\hat{y}\|x)$	At fault?
Jane visits Africa in September.	Jane visited Africa last September.	2×10^{-10}	1×10^{-10}	B R B R R ⋮

[https://postimg.cc/image/nxz4orzxr/]

Through this process, you can carry out error analysis to figure out what fraction of errors are due to *beam search* versus the *RNN model*. For every example in your dev sets where the algorithm gives a much worse output than the human translation, you can try to ascribe the error to either the search algorithm or to the objective function, or to the RNN model that generates the objective function that beam search is supposed to be maximizing. If you find that beam search is responsible for a lot of errors, then maybe it is worth increasing the beam width. Whereas in contrast, if you find that the RNN model is at fault, 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.

Sequence models & Attention mechanism: Attention Model Intuition

For most of this week, we have been looking at seq2seq models. A simple modification to this architecture makes it much more powerful, lets take a look.

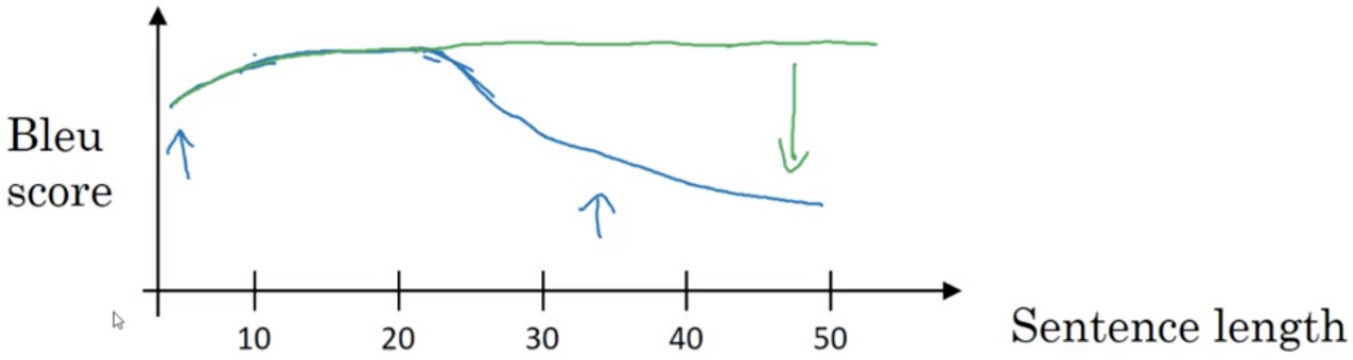
The problem of long sequences

Given a very long French sentence, the encoder in a seq2seq network must read in the whole sentence and then, essentially memorize it by storing its learned representation the activations values. Then the decoder network will then generate the English translation.

x : "Jane went to Africa last September and enjoyed the culture and met many wonderful people; she came b

> Example output (translated sentence)

Now, the way a human translator would translate this sentence is *not* by reading and memorizing the entire sentence before beginning translation. Instead, the human translator would read the sentence bit-by-bit, translating words as they go, and paying special attention to certain parts of the input sentence when deciding what the ouput sentence should be. For the seq2seq architecture we introduced earlier, we finde that it works quite well for short sentences, but for very long sentences (maybe longer than 30 or 40 words) performance drops.



[https://postimg.cc/image/4meyki473/]

> Short sentences are just hard to translate in general due to the lack of context. For long sentences, the vanilla seq2seq model doesn't do well because it's difficult to for the network to memorize a very long sentence. Blue line: seq2seq architectures without attention, Green line: seq2seq architectures with attention.

In this and the next lecture, you'll see the **Attention Model** which translates maybe a bit more like humans might, by looking at part of the sentence at a time. With an Attention model, machine translation systems performance stabilizes across sentence lengths. This is mostly due to the fact that, without attention, we are inadvertantly measuring the ability of a neural network to *memorize* a long sentence which isn't what is most important.

Attention model intuition

Lets build our intuition for the **attention model** with a simple example:

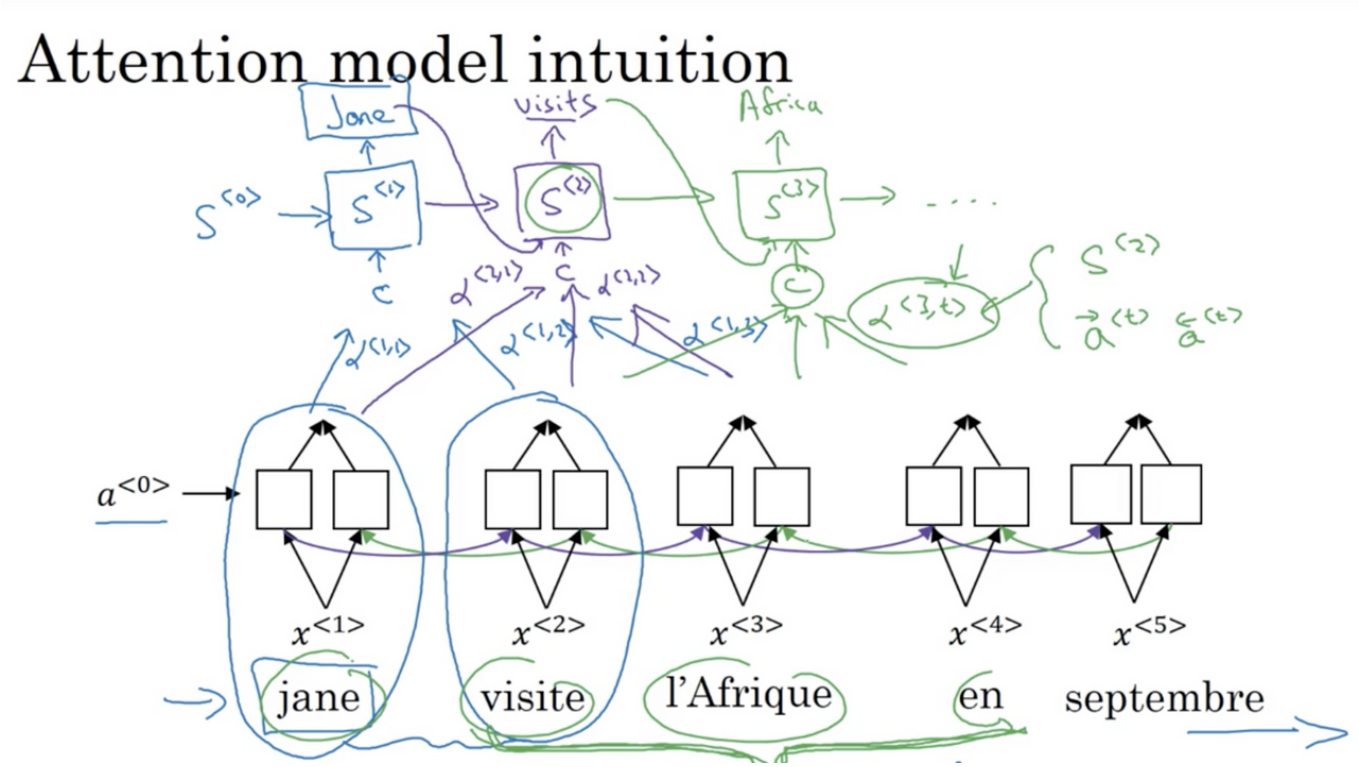
$$x : \text{"Jane visite l'Afrique en septembre"}$$

> Attention was first introduced here [https://arxiv.org/abs/1409.0473].

Note that attention is typically more useful for longer sequences, but for the purposes of illustration we will use a rather short one.

Say that we use a bi-directional RNN to compute some sort of rich feature set for a given input. Lets introduce a new notation: $\alpha^{<i,j>}$ can be thought of as an attention measure, e.g., when we are looking to produce the translated word i in the output sequence, how much *attention* should we pay to the word j in the input sequence? Broadly speaking, our decoder takes these measures of attention, along with the output from the encoder network to compute a new input that it uses when

determining its outputs. More specifically, $\alpha^{<i,j>}$ is influenced by the forward and backward activations of the encoder network at timestep i and the output from the previous timestep of the decoder network, $S^{<t-1>}$.



[https://postimg.cc/image/q3zv301j3/]

> Note, we have denoted the activations of the decoder network as $S^{<t>}$ to avoid confusion.

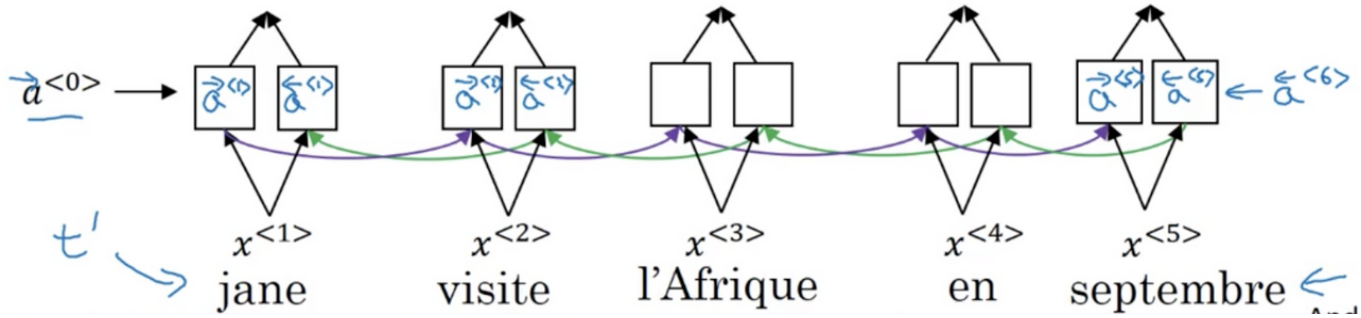
The key intuition is that the decoder network marches away, generating an output at every timestep. The attention weights help the decoder determine what information to pay attention to in the encoders output, as opposed to using all information in the learned representation from the entire input sequence.

Sequence models & Attention mechanism: Attention Model

In the last lecture, we saw how the attention model allows a neural network to pay attention to only part of an input sentence while it's generating a translation, much like a human translator might. Let's now formalize that intuition, and see how we might implement this attention model ourselves.

Attention model

Similar to the last lecture, lets assume we have a bi-directional RNN (be it LSTM or GRU) that we use to compute features for the input sequence.



[https://postimg.cc/image/qt5scfldr/]

To simplify the notation, at every timestep we are going to denote the activations for both the forward and backward recurrent networks as $a^{<t>}$, such that:

$$a^{<t'>} = (\overrightarrow{a}^{<t'>}, \overleftarrow{a}^{<t'>})$$

We also have a second RNN (in the forward direction only) with states $S^{<t>}$ which takes as input some context c, where c depends on the attention parameters ($\alpha^{<1,1>}, \alpha^{<1,2>}, etc$). These α parameters tell us how much the context c should depend on the learned features across the timesteps, which is a weighted sum. More formally:

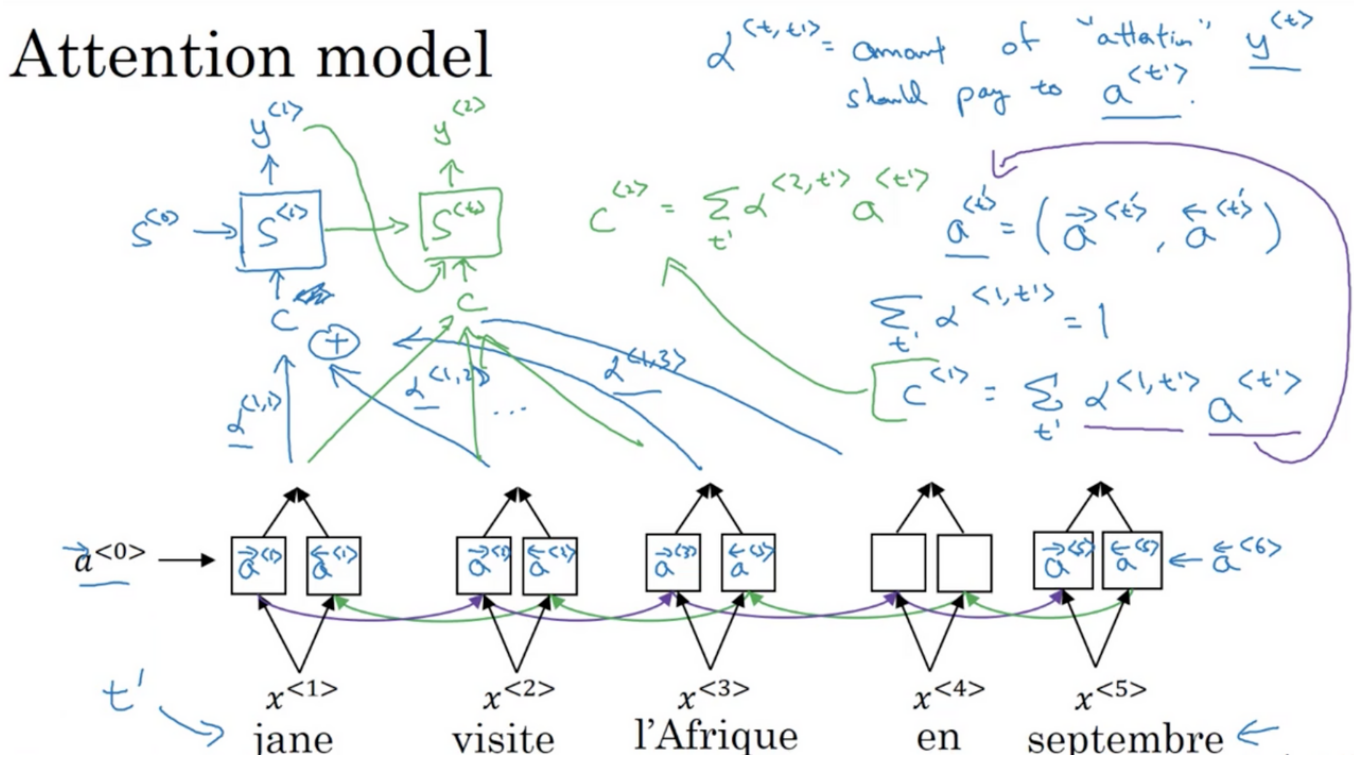
$$\sum_{t'} \alpha^{<1,t'>} = 1$$

$$c^{<1>} = \sum_{t'} \alpha^{<1,t'>} a^{<t'>}$$

In plain english, $\alpha^{<t,t'>}$ is the amount of "attention" that $y^{<t>}$ should pay to $a^{<t'>}$. We compute the context for the second timestep, $c^{<2>}$ in a similar manner:

$$c^{<2>} = \sum_{t'} \alpha^{<2,t'>} a^{<t'>}$$

And so forth and so on.



[https://postimg.cc/image/obu155m1r/]

The only thing left to define is how we *actually* compute the attention weights, $\alpha^{<t,t'>}$

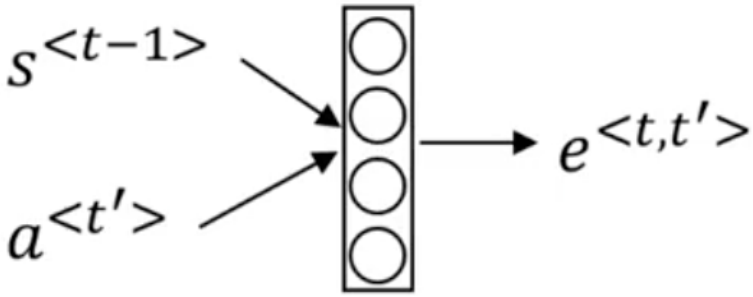
Computing attentions $\alpha^{<t,t'>}$

Let us first present the formula and then explain it:

$$\alpha^{<t,t'>} = \frac{exp(e^{<t,t'>})}{\sum_{t'=1}^{T_x} exp(e^{<t,t'>})}$$

> It is helpful to keep in mind that $\alpha^{<t,t'>}$ is the amount of "attention" that $y^{<t>}$ should pay to $a^{<t'>}$

We first compute $e^{<t,t'>}$ and then use what is essentially a softmax over all t' to guarantee that $\alpha^{<t,t'>}$ sums to zero over all t'. But how do we compute $e^{<t,t'>}$? Typically we use a small neural network that looks something like the following:



[https://postimg.cc/image/whc33auv3/]

The intuition is, if you want to decide how much attention to pay to the activation of t', we should depend on the hidden state activation from the previous time step, $S^{<t-1>}$ and the learned features of the word at t', $a^{<t'>}$. What we don't know is the exact function which takes in ($S^{<t-1>}$, $a^{<t'>}$) and maps it to $e^{<t,t'>}$. Therefore, we use a small neural network to learn this function for us.

> The network MUST be small as we need to use it to make a prediction at every timestep t.
>
> Further reading: Neural machine translation by jointly learning to align and translate [https://arxiv.org/abs/1409.0473] and Show, attend and tell: Neural image caption generation with visual attention [https://arxiv.org/pdf/1502.03044.pdf].

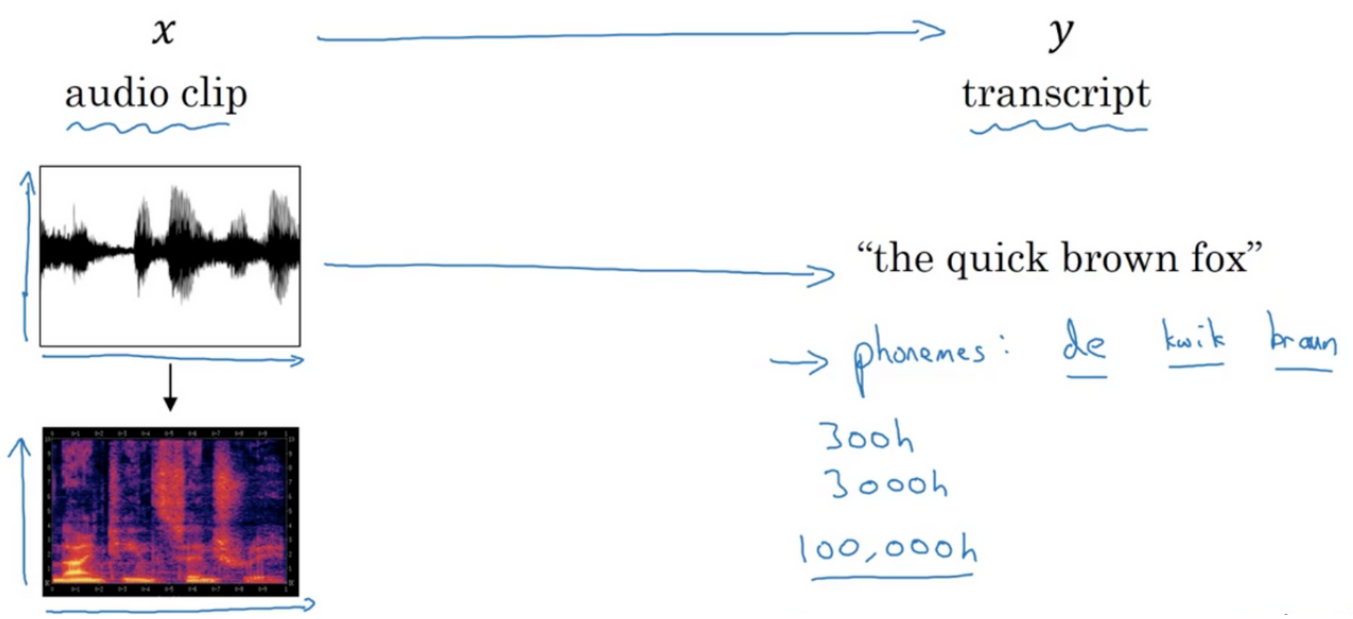
Speech recognition - Audio data: Speech recognition

One of the most exciting developments with **sequence-to-sequence models** has been the rise of very accurate speech recognition. We will spend the last two lectures building a sense of how these sequence-to-sequence models are applied to audio data, such as the speech.

Speech recognition problem

In speech recognition, produce a transcript y given an audio clip, x. The audio transcript is generated from a microphone (i.e., someone speaks into a microphone, speech, like all sound, is a pressure wave which the microphone pics up on and generates a spectrum).

In truth, even the human brain doesn't process *raw sound*. We typically perform a series of pre-processing steps, eventually generating a spectrogram, a 3D representation of sound (x: time, y: frequency, z: amplitude)



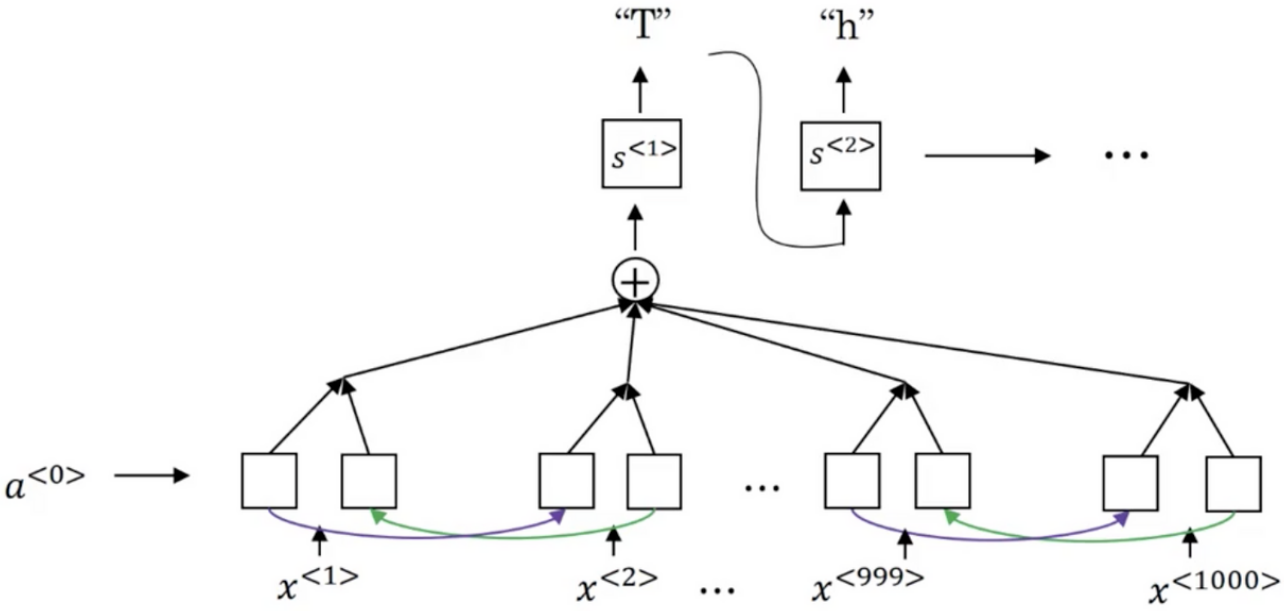
[https://postimg.cc/image/eh2vywn8f/]

> The human ear performs something similar to this preprocessing.

Traditionally, speech recognition was solved by first learning to classify **phonemes** (typically we expensive feature engineering), the individual units of speech. However, with more powerful end-to-end architectures we are finding that explicit phoneme recognition is no longer necessary, or even a good idea. With these arcitectures, a large amount of data is especially important. A large academic dataset may be around 3000h of speech, while commercial systems (e.g. Alex) may be trained on datasets containing as many as 100,000h of speech.

Attention model for speech recognition

One way to build a speech recognition system is to use the seq2seq model with attention that we explored previously, where the audio clip (divided into small timeframes) is the input and the audio transcript is the output



[https://postimg.cc/image/yboxl07kv/]

CTC cost for speech recognition

Lets say our audio clip is someone saying the sentence "the quick brown fox". Typically in speech recognition, the length of our input sequence is much much larger than the length of our output sequence.

> For example, a 10 second, 100 Hz audio clip becomes a 1000 inputs.

Connectionist temporal classification (CTC) allows us use a bi-directional RNN where $T_x == T_y$, by allowing the model to predict both repeated characters and "blanks". For our examples, the predicted transcript might look like:

$$\text{ttt \_ h \_ eee \_ \_ \_ \_ <SPACE> \_ \_ \_ qqq \_ \_ ...}$$

> Where _ denotes a "blank".

The basic rule is to *collapse repeated characters not separated by a "blank"*.

> This idea was originally introduced here [http://www.cs.toronto.edu/~graves/icml_2006.pdf]

Summary

In this lecture we tried to get a rough sense of how speech recognition models work. We saw:

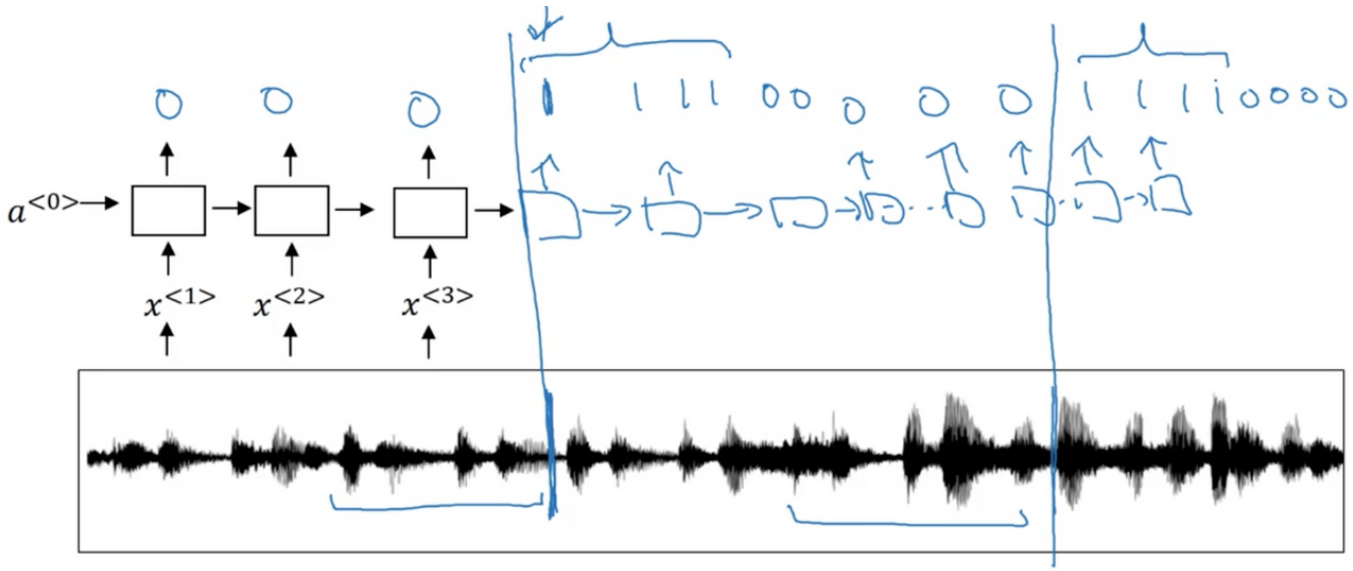
- two methods for building a speech recognition system (both involved bi-directional RNNs): a **seq2seq model with attention** and a **CTC model**.
- that deep learning has had a dramatic impact of the viability of commercial speech recognition systems.
- building effective speech recognition system is still requires a very significant effort and a very large data set.

Speech recognition - Audio data: Trigger word detection

Trigger word detection systems are used to identify **wake** (or trigger) **words** (e.g. "Hey Google", "Alexa"), typically for digital assistants like Alexa, Google Home and Siri.

The literature on trigger word detection is still evolving, and there is not widespread consensus or a universally agreed upon algorithm for trigger detection. We are just going to look at one example.

Our task is to take an audio clip, possibly perform preprocessing to compute a spectrogram and then the features that we will eventually pass to an RNN, and predict at which timesteps the wake word was uttered. One strategy, is to have the neural network output the label 0 for all timesteps before a wake word is mentioned and then 1 directly after it is mentioned. The slight problem with this is that our training set becomes very unbalanced (many more 0's than 1's). Instead, we typically have the model predict a few 1's for the timesteps that come directly after the wake word was mentioned.



[https://postimg.cc/image/7dv0j8s2n/]