kaddynator Update Readme.md ... on 1 Aug 2019 History

..	
Images	4 years ago
Readme.md	2 years ago

Sequence Models

This is the fifth and final course of the deep learning specialization at Coursera which is moderated by deeplearning.ai. The course is taught by Andrew Ng.

Table of contents

Course summary

Here are the course summary as its given on the course link:

> This course will teach you how to build models for natural language, audio, and other sequence data. Thanks to deep learning, sequence algorithms are working far better than just two years ago, and this is enabling numerous exciting applications in speech recognition, music synthesis, chatbots, machine translation, natural language understanding, and many others.
>
> You will:
>
> - Understand how to build and train Recurrent Neural Networks (RNNs), and commonly-used variants such as GRUs and LSTMs.
> - Be able to apply sequence models to natural language problems, including text synthesis.
> - Be able to apply sequence models to audio applications, including speech recognition and music synthesis.
>
> This is the fifth and final course of the Deep Learning Specialization.

Recurrent Neural Networks

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

Why sequence models

- Sequence Models like RNN and LSTMs have greatly transformed learning on sequences in the past few years.
- Examples of sequence data in applications:
    - Speech recognition (**sequence to sequence**):
        - X: wave sequence
        - Y: text sequence
    - Music generation (**one to sequence**):
        - X: nothing or an integer
        - Y: wave sequence
    - Sentiment classification (**sequence to one**):
        - X: text sequence
        - Y: integer rating from one to five
    - DNA sequence analysis (**sequence to sequence**):
        - X: DNA sequence
        - Y: DNA Labels

- Machine translation (**sequence to sequence**):
  - X: text sequence (in one language)
  - Y: text sequence (in other language)
- Video activity recognition (**sequence to one**):
  - X: video frames
  - Y: label (activity)
- Name entity recognition (**sequence to sequence**):
  - X: text sequence
  - Y: label sequence
  - Can be used by seach engines to index different type of words inside a text.

- All of these problems with different input and output (sequence or not) can be addressed as supervised learning with label data X, Y as the training set.

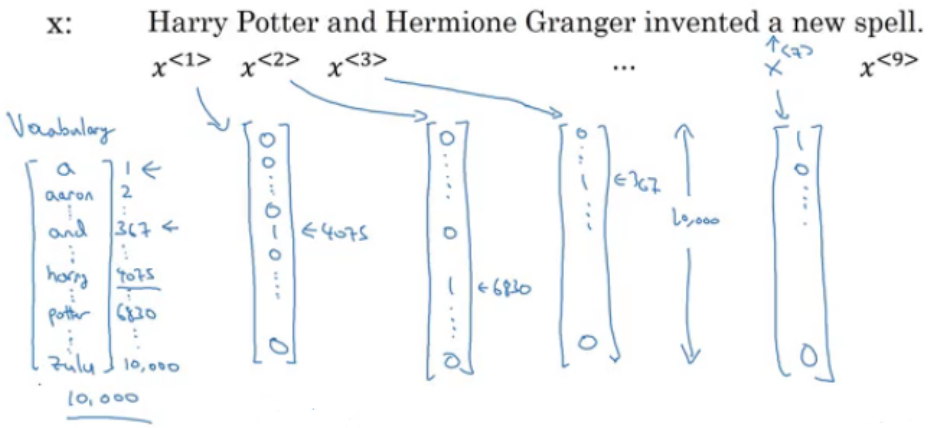
Notation

- In this section we will discuss the notations that we will use through the course.
- **Motivating example**:
  - Named entity recognition example:
    - X: "Harry Potter and Hermoine Granger invented a new spell."
    - Y: 1 1 0 1 1 0 0 0 0
    - Both elements has a shape of 9. 1 means its a name, while 0 means its not a name.
- We will index the first element of x by $x^{<1>}$, the second $x^{<2>}$ and so on.
  - $x^{<1>}$ = Harry
  - $x^{<2>}$ = Potter
- Similarly, we will index the first element of y by $y^{<1>}$, the second $y^{<2>}$ and so on.
  - $y^{<1>}$ = 1
  - $y^{<2>}$ = 1
- T_x is the size of the input sequence and T_y is the size of the output sequence.
  - $T_x = T_y = 9$ in the last example although they can be different in other problems.
- $x^{(i)<t>}$ is the element t of the sequence of input vector i. Similarly $y^{(i)<t>}$ means the t-th element in the output sequence of the i training example.
- $T_x^{(i)}$ the input sequence length for training example i. It can be different across the examples. Similarly for $T_y^{(i)}$ will be the length of the output sequence in the i-th training example.
- **Representing words**:
  - We will now work in this course with **NLP** which stands for natural language processing. One of the challenges of NLP is how can we represent a word?

  i. We need a **vocabulary** list that contains all the words in our target sets.
    - Example:
      - [a ... And ... Harry ... Potter ... Zulu]
      - Each word will have a unique index that it can be represented with.
      - The sorting here is in alphabetical order.
    - Vocabulary sizes in modern applications are from 30,000 to 50,000. 100,000 is not uncommon. Some of the bigger companies use even a million.
    - To build vocabulary list, you can read all the texts you have and get m words with the most occurrence, or search online for m most occurrent words.

  ii. Create a **one-hot encoding** sequence for each word in your dataset given the vocabulary you have created.
    - While converting, what if we meet a word thats not in your dictionary?
    - We can add a token in the vocabulary with name `<UNK>` which stands for unknown text and use its index for your one-hot vector.

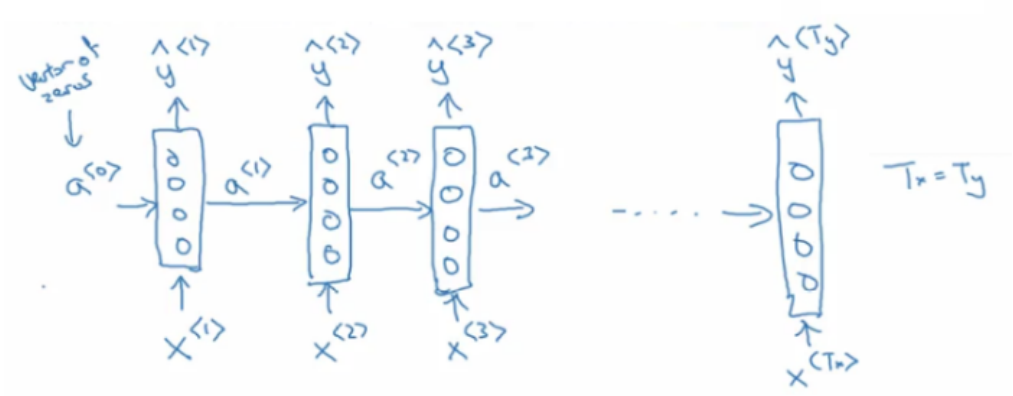
- Full example:



- The goal is given this representation for x to learn a mapping using a sequence model to then target output y as a supervised learning problem.

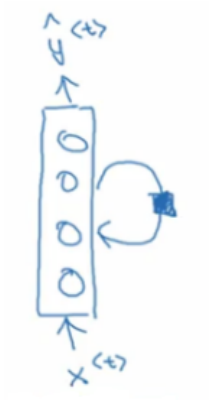
Recurrent Neural Network Model

- Why not to use a standard network for sequence tasks? There are two problems:
  - Inputs, outputs can be different lengths in different examples.
    - This can be solved for normal NNs by paddings with the maximum lengths but it's not a good solution.
  - Doesn't share features learned across different positions of text/sequence.
    - Using a feature sharing like in CNNs can significantly reduce the number of parameters in your model. That's what we will do in RNNs.
- Recurrent neural network doesn't have either of the two mentioned problems.
- Lets build a RNN that solves **name entity recognition** task:

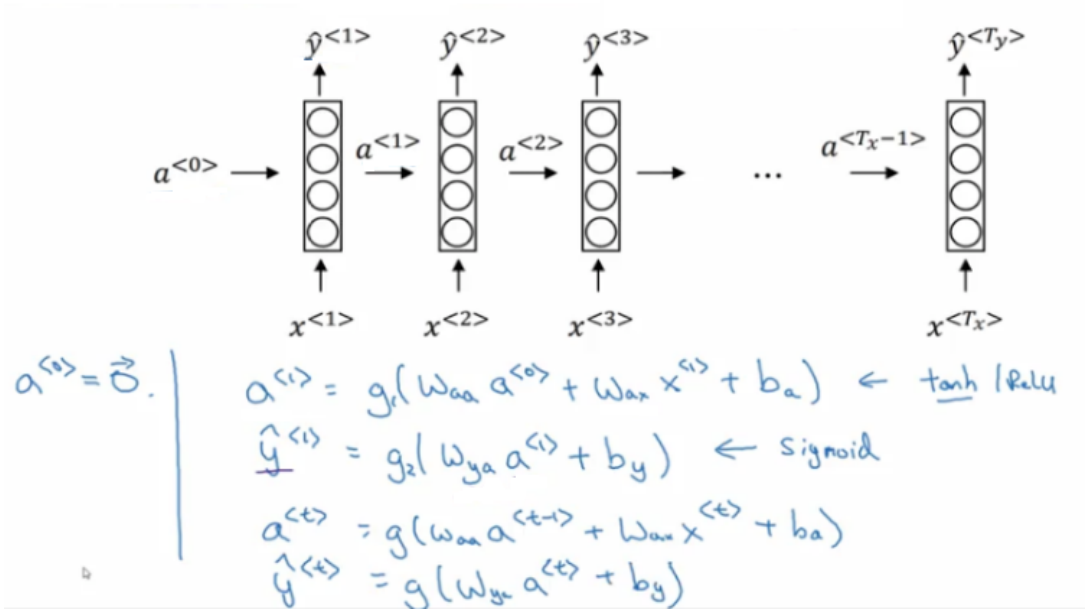


  - In this problem $T_x = T_y$. In other problems where they aren't equal, the RNN architecture may be different.
  - $a^{<0>}$ is usually initialized with zeros, but some others may initialize it randomly in some cases.
  - There are three weight matrices here: W_{ax}, W_{aa}, and W_{ya} with shapes:
    - W_{ax}: (NoOfHiddenNeurons, n_x)
    - W_{aa}: (NoOfHiddenNeurons, NoOfHiddenNeurons)
    - W_{ya}: (n_y, NoOfHiddenNeurons)
- The weight matrix W_{aa} is the memory the RNN is trying to maintain from the previous layers.

- A lot of papers and books write the same architecture this way:



  - It's harder to interpreter. It's easier to roll this drawings to the unrolled version.
- In the discussed RNN architecture, the current output $\hat{y}^{<t>}$ depends on the previous inputs and activations.
- Let's have this example 'He Said, "Teddy Roosevelt was a great president"'. In this example Teddy is a person name but we know that from the word **president** that came after Teddy not from **He** and **said** that were before it.
- So limitation of the discussed architecture is that it can not learn from elements later in the sequence. To address this problem we will later discuss **Bidirectional RNN** (BRNN).
- Now let's discuss the forward propagation equations on the discussed architecture:



  - The activation function of a is usually tanh or ReLU and for y depends on your task choosing some activation functions like sigmoid and softmax. In name entity recognition task we will use sigmoid because we only have two classes.
- In order to help us develop complex RNN architectures, the last equations needs to be simplified a bit.
- **Simplified RNN notation**:

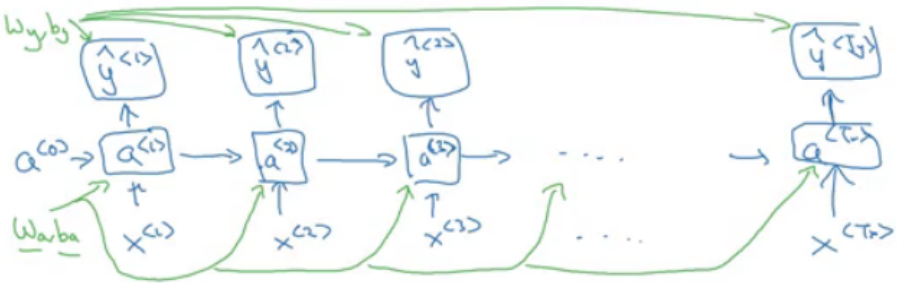
$a^{<t>} = g(W_{aa}a^{<t>} + W_{ax}x^{<t>} + b_a)$	$a^{<t>} = g(W_a[a^{<t-1>}, x^{<t>}] + b_a)$ $W_a = [W_{aa}; W_{ax}]$ $[a^{<t-1>}, x^{<t>}] = \begin{bmatrix} a^{<t-1>} \\ x^{<t>} \end{bmatrix}$
$\hat{y}^{<t>} = g(W_{ya}a^{<t>} + b_y)$	$\hat{y}^{<t>} = g(W_y a^{<t>} + b_y)$

  - w_a is w_{aa} and w_{ax} stacked horizontally.

- $[a^{<t-1>}, x^{<t>}]$ is $a^{<t-1>}$ and $x^{<t>}$ stacked vertically.
- w_a shape: (NoOfHiddenNeurons, NoOfHiddenNeurons + n_x)
- $[a^{<t-1>}, x^{<t>}]$ shape: (NoOfHiddenNeurons + n_x, 1)

Backpropagation through time

- Let's see how backpropagation works with the RNN architecture.
- Usually deep learning frameworks do backpropagation automatically for you. But it's useful to know how it works in RNNs.
- Here is the graph:

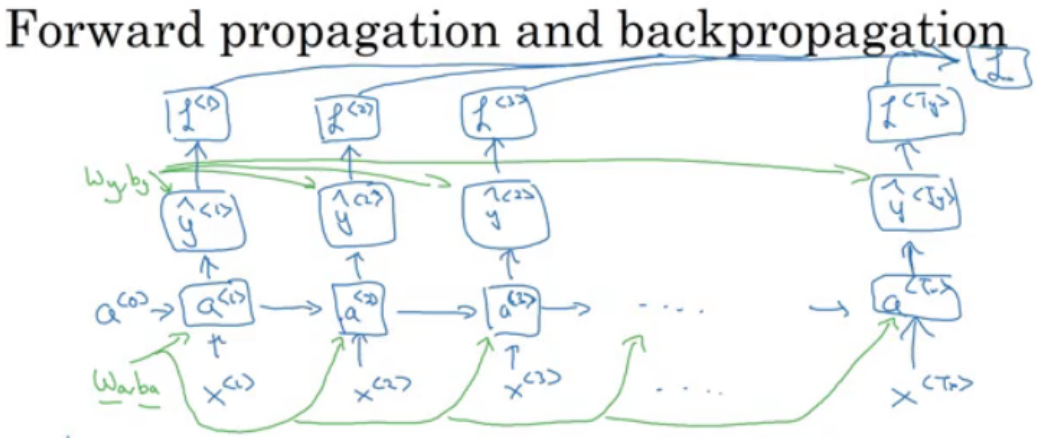


  - Where w_a, b_a, w_y, and b_y are shared across each element in a sequence.
- We will use the cross-entropy loss function:

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

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

  - Where the first equation is the loss for one example and the loss for the whole sequence is given by the summation over all the calculated single example losses.
- Graph with losses:

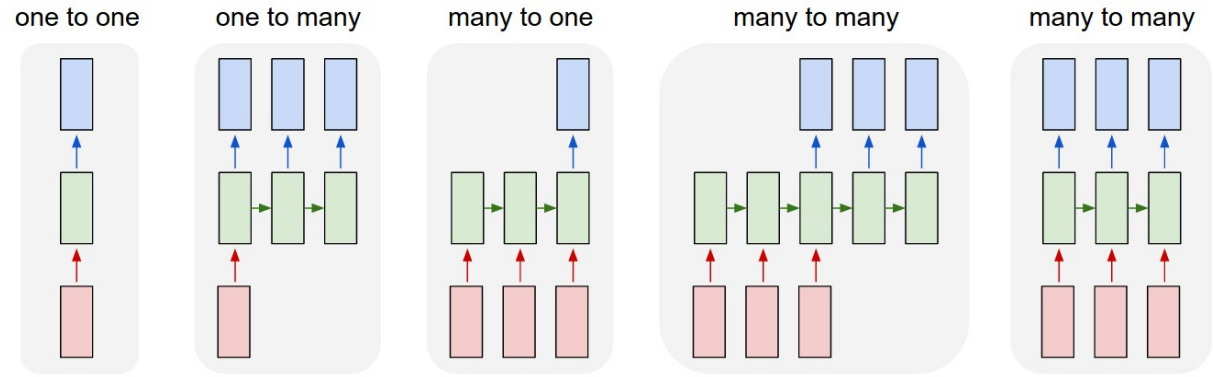


- The backpropagation here is called **backpropagation through time** because we pass activation a from one sequence element to another like backwards in time.

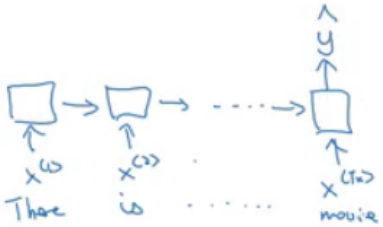
Different types of RNNs

- So far we have seen only one RNN architecture in which T_x equals T_Y. In some other problems, they may not equal so we need different architectures.

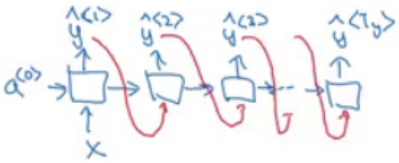
- The ideas in this section was inspired by Andrej Karpathy blog. Mainly this image has all types:



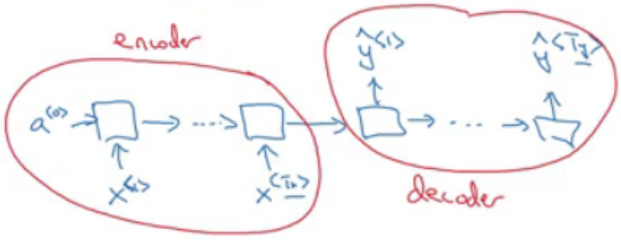
- The architecture we have described before is called **Many to Many**.
- In sentiment analysis problem, X is a text while Y is an integer that rangers from 1 to 5. The RNN architecture for that is **Many to One** as in Andrej Karpathy image.



- A **One to Many** architecture application would be music generation.

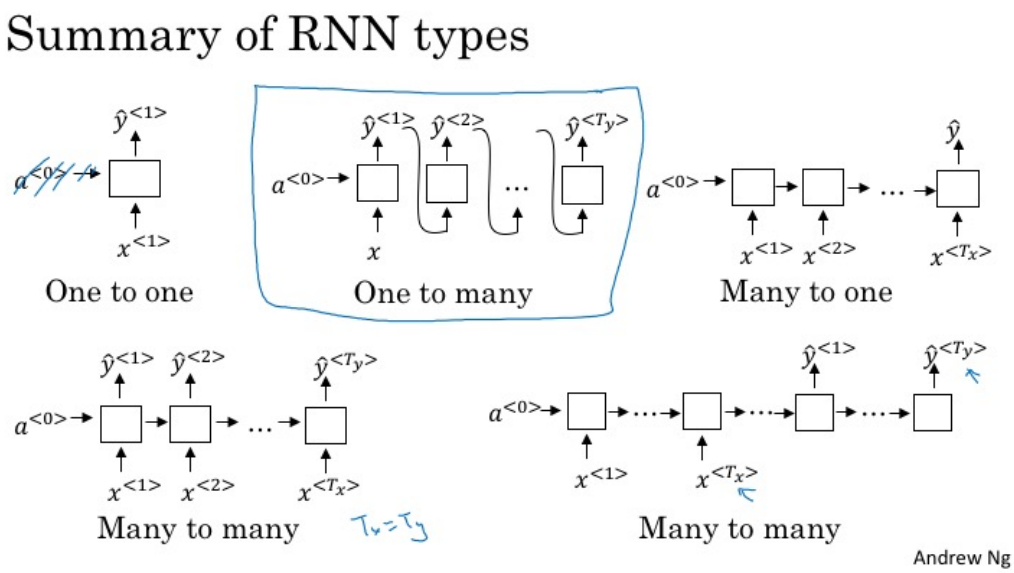


  - Note that starting the second layer we are feeding the generated output back to the network.
- There are another interesting architecture in **Many To Many**. Applications like machine translation inputs and outputs sequences have different lengths in most of the cases. So an alternative *Many To Many* architecture that fits the translation would be as follows:



  - There are an encoder and a decoder parts in this architecture. The encoder encodes the input sequence into one matrix and feed it to the decoder to generate the outputs. Encoder and decoder have different weight matrices.

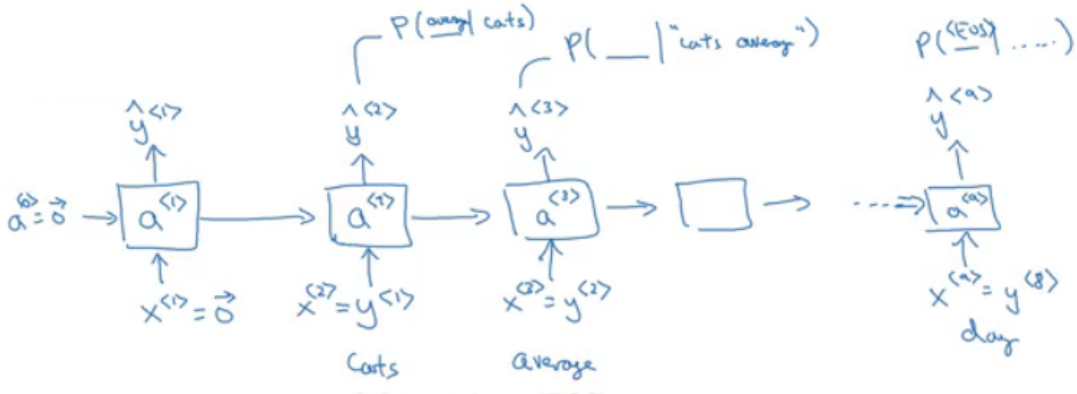
- Summary of RNN types:



- There is another architecture which is the **attention** architecture which we will talk about in chapter 3.

Language model and sequence generation

- RNNs do very well in language model problems. In this section, we will build a language model using RNNs.
- **What is a language model**
  - Let's say we are solving a speech recognition problem and someone says a sentence that can be interpreted into to two sentences:
    - The apple and **pair** salad
    - The apple and **pear** salad
  - **Pair** and **pear** sounds exactly the same, so how would a speech recognition application choose from the two.
  - That's where the language model comes in. It gives a probability for the two sentences and the application decides the best based on this probability.
- The job of a language model is to give a probability of any given sequence of words.
- **How to build language models with RNNs?**
  - The first thing is to get a **training set**: a large corpus of target language text.
  - Then tokenize this training set by getting the vocabulary and then one-hot each word.
  - Put an end of sentence token `<EOS>` with the vocabulary and include it with each converted sentence. Also, use the token `<UNK>` for the unknown words.
- Given the sentence "Cats average 15 hours of sleep a day. `<EOS>` "
  - In training time we will use this:



  - The loss function is defined by cross-entropy loss:

$$\mathcal{L}(\hat{y}^{<t>}, y^{<t>}) = -\sum_i y_i^{<t>} \log \hat{y}_i^{<t>}$$

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

    - `i` is for all elements in the corpus, `t` - for all timesteps.
- To use this model:

i. For predicting the chance of **next word**, we feed the sentence to the RNN and then get the final $y^{\hat{}<t>}$ hot vector and sort it by maximum probability.

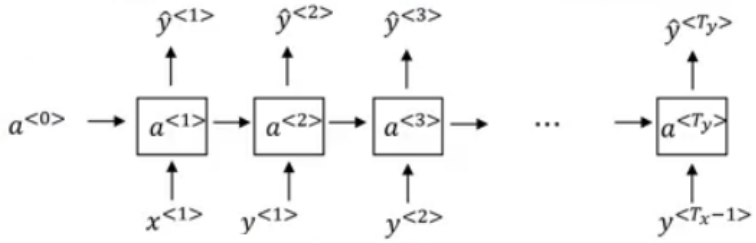
ii. For taking the **probability of a sentence**, we compute this:

- $p(y^{<1>}, y^{<2>}, y^{<3>}) = p(y^{<1>}) * p(y^{<2>} \mid y^{<1>}) * p(y^{<3>} \mid y^{<1>}, y^{<2>})$
- This is simply feeding the sentence into the RNN and multiplying the probabilities (outputs).

Sampling novel sequences

- After a sequence model is trained on a language model, to check what the model has learned you can apply it to sample novel sequence.
- Lets see the steps of how we can sample a novel sequence from a trained sequence language model:

i. Given this model:



ii. We first pass $a^{<0>}$ = zeros vector, and $x^{<1>}$ = zeros vector.

iii. Then we choose a prediction randomly from distribution obtained by $\hat{y}^{<1>}$. For example it could be "The".

- In numpy this can be implemented using: `numpy.random.choice(...)`
- This is the line where you get a random beginning of the sentence each time you sample run a novel sequence.

iv. We pass the last predicted word with the calculated $a^{<1>}$

v. We keep doing 3 & 4 steps for a fixed length or until we get the `<EOS>` token.

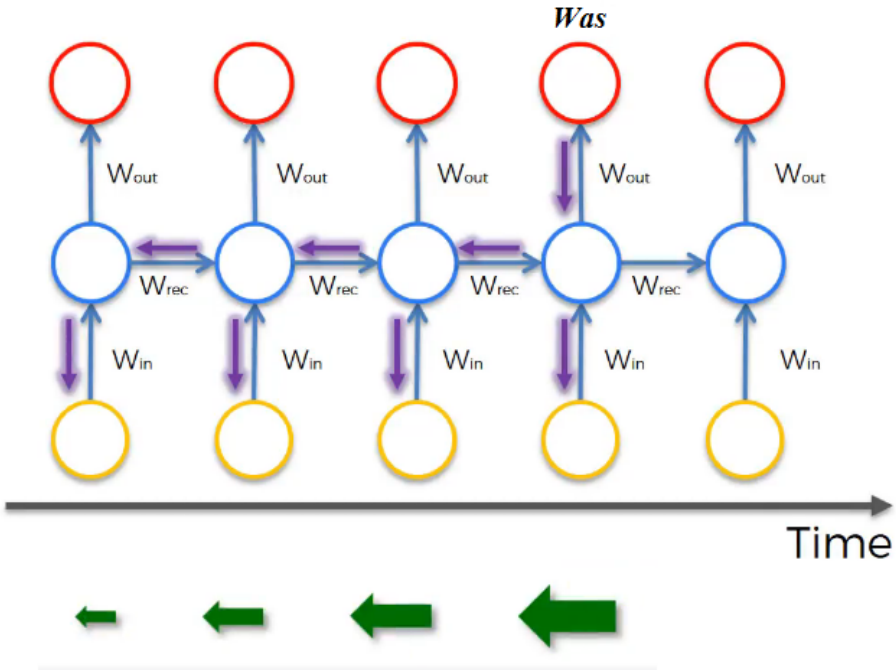
vi. You can reject any `<UNK>` token if you mind finding it in your output.

- So far we have to build a word-level language model. It's also possible to implement a **character-level** language model.
- In the character-level language model, the vocabulary will contain `[a-zA-Z0-9]` , punctuation, special characters and possibly token.
- Character-level language model has some pros and cons compared to the word-level language model
  - Pros:
    a. There will be no `<UNK>` token - it can create any word.
  - Cons:
    a. The main disadvantage is that you end up with much longer sequences.
    b. Character-level language models are not as good as word-level language models at capturing long range dependencies between how the the earlier parts of the sentence also affect the later part of the sentence.
    c. Also more computationally expensive and harder to train.
- The trend Andrew has seen in NLP is that for the most part, a word-level language model is still used, but as computers get faster there are more and more applications where people are, at least in some special cases, starting to look at more character-level models. Also, they are used in specialized applications where you might need to deal with unknown words or other vocabulary words a lot. Or they are also used in more specialized applications where you have a more specialized vocabulary.

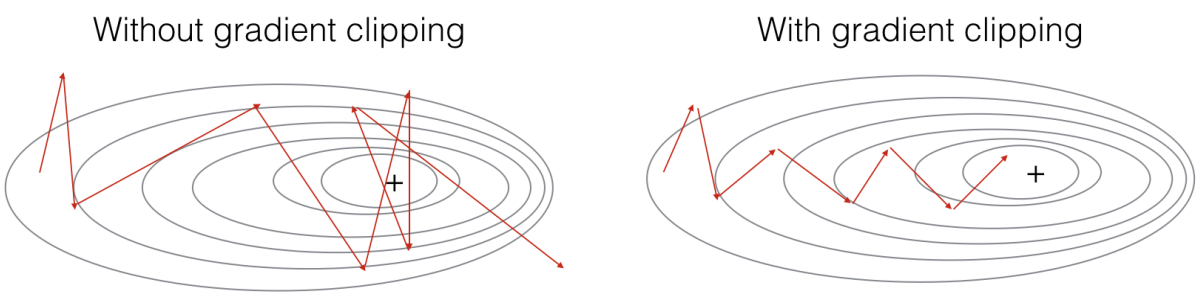
Vanishing gradients with RNNs

- One of the problems with naive RNNs that they run into **vanishing gradient** problem.
- An RNN that process a sequence data with the size of 10,000 time steps, has 10,000 deep layers which is very hard to optimize.
- Let's take an example. Suppose we are working with language modeling problem and there are two sequences that model tries to learn:
  - "The **cat**, which already ate ..., **was** full"
  - "The **cats**, which already ate ..., **were** full"
  - Dots represent many words in between.

- What we need to learn here that "was" came with "cat" and that "were" came with "cats". The naive RNN is not very good at capturing very long-term dependencies like this.
- As we have discussed in Deep neural networks, deeper networks are getting into the vanishing gradient problem. That also happens with RNNs with a long sequence size.



- For computing the word "was", we need to compute the gradient for everything behind. Multiplying fractions tends to vanish the gradient, while multiplication of large number tends to explode it.
    - Therefore some of your weights may not be updated properly.
- In the problem we descried it means that its hard for the network to memorize "was" word all over back to "cat". So in this case, the network won't identify the singular/plural words so that it gives it the right grammar form of verb was/were.
- The conclusion is that RNNs aren't good in **long-term dependencies**.
- > In theory, RNNs are absolutely capable of handling such “long-term dependencies.” A human could carefully pick parameters for them to solve toy problems of this form. Sadly, in practice, RNNs don’t seem to be able to learn them. http://colah.github.io/posts/2015-08-Understanding-LSTMs/
- *Vanishing gradients* problem tends to be the bigger problem with RNNs than the *exploding gradients* problem. We will discuss how to solve it in next sections.
- Exploding gradients can be easily seen when your weight values become `NaN`. So one of the ways solve exploding gradient is to apply **gradient clipping** means if your gradient is more than some threshold - re-scale some of your gradient vector so that is not too big. So there are cliped according to some maximum value.

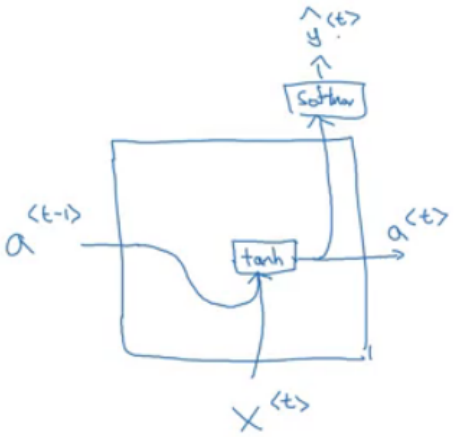


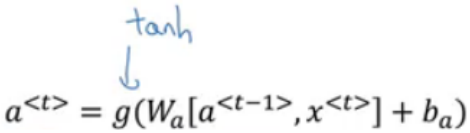
- **Extra**:
    - Solutions for the Exploding gradient problem:
        - Truncated backpropagation.
            - Not to update all the weights in the way back.
            - Not optimal. You won't update all the weights.

        - Gradient clipping.
    - Solution for the Vanishing gradient problem:
        - Weight initialization.
            - Like He initialization.
        - Echo state networks.
        - Use LSTM/GRU networks.
            - Most popular.
            - We will discuss it next.

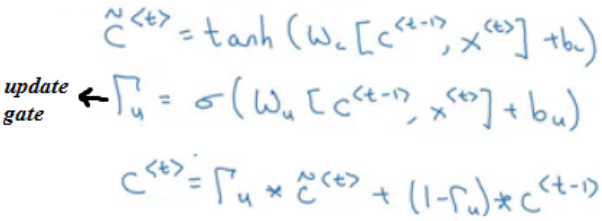
Gated Recurrent Unit (GRU)

- GRU is an RNN type that can help solve the vanishing gradient problem and can remember the long-term dependencies.
- The basic RNN unit can be visualized to be like this:





- We will represent the GRU with a similar drawings.
- Each layer in **GRUs** has a new variable c which is the memory cell. It can tell to whether memorize something or not.
- In GRUs, $C^{<t>} = a^{<t>}$
- Equations of the GRUs:

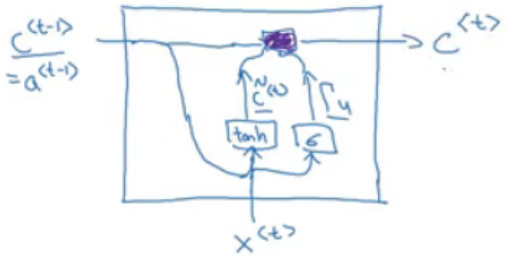


    - The update gate is between 0 and 1
        - To understand GRUs imagine that the update gate is either 0 or 1 most of the time.
    - So we update the memory cell based on the update cell and the previous cell.
- Lets take the cat sentence example and apply it to understand this equations:
    - Sentence: "The **cat**, which already ate ....., **was** full"
    - We will suppose that U is 0 or 1 and is a bit that tells us if a singular word needs to be memorized.
    - Splitting the words and get values of C and U at each place:

Word	Update gate(U)	Cell memory (C)
The	0	val
cat	1	new_val

Word	Update gate(U)	Cell memory (C)
which	0	new_val
already	0	new_val
...	0	new_val
was	1 (I don't need it anymore)	newer_val
full	..	..

- Drawing for the GRUs



  - Drawings like in http://colah.github.io/posts/2015-08-Understanding-LSTMs/ is so popular and makes it easier to understand GRUs and LSTMs. But Andrew Ng finds it's better to look at the equations.

- Because the update gate U is usually a small number like 0.00001, GRUs doesn't suffer the vanishing gradient problem.
  - In the equation this makes $C^{<t>} = C^{<t-1>}$ in a lot of cases.
- Shapes:
  - $a^{<t>}$ shape is (NoOfHiddenNeurons, 1)
  - $c^{<t>}$ is the same as $a^{<t>}$
  - $c^{\sim<t>}$ is the same as $a^{<t>}$
  - $u^{<t>}$ is also the same dimensions of $a^{<t>}$
- The multiplication in the equations are element wise multiplication.
- What has been descried so far is the Simplified GRU unit. Let's now describe the full one:
  - The full GRU contains a new gate that is used with to calculate the candidate C. The gate tells you how relevant is $C^{<t-1>}$ to $C^{<t>}$
  - Equations:

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

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

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

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

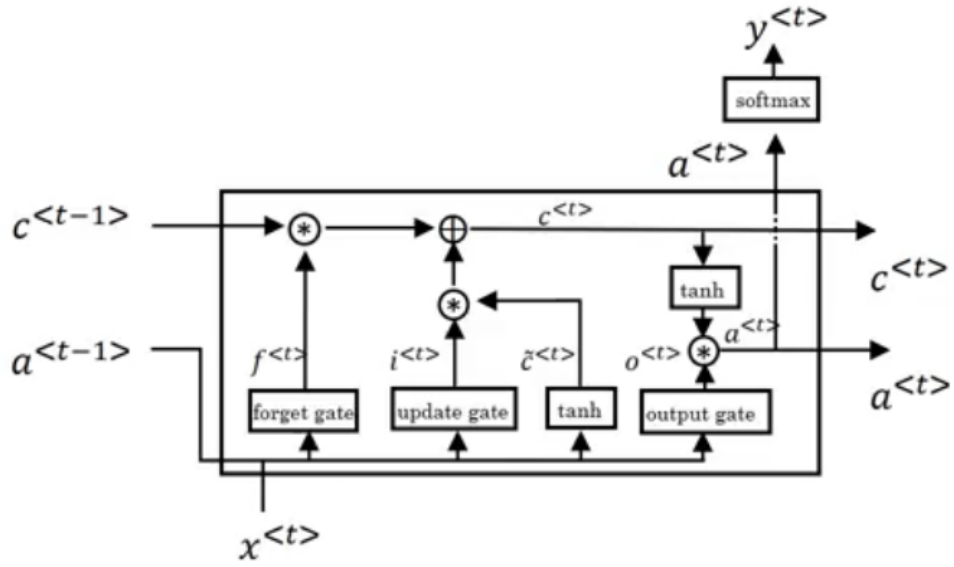
  - Shapes are the same
- So why we use these architectures, why don't we change them, how we know they will work, why not add another gate, why not use the simpler GRU instead of the full GRU; well researchers has experimented over years all the various types of these architectures with many many different versions and also addressing the vanishing gradient problem. They have found that full GRUs are one of the best RNN architectures to be used for many different problems. You can make your design but put in mind that GRUs and LSTMs are standards.

Long Short Term Memory (LSTM)

- LSTM - the other type of RNN that can enable you to account for long-term dependencies. It's more powerful and general than GRU.
- In LSTM , $C^{<t>}$!= $a^{<t>}$
- Here are the equations of an LSTM unit:

$$\tilde{c}^{<t>} = \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c)$$
$$\Gamma_u = \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u) \quad \text{(update)}$$
$$\Gamma_f = \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f) \quad \text{(forget)}$$
$$\Gamma_o = \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o) \quad \text{(output)}$$
$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>}$$
$$a^{<t>} = \Gamma_o * \tanh c^{<t>}$$

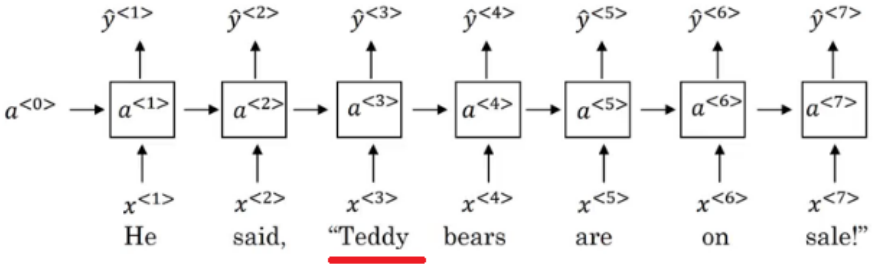
- In GRU we have an update gate `u` , a relevance gate `r` , and a candidate cell variables $C^{\sim<t>}$ while in LSTM we have an update gate `u` (sometimes it's called input gate I), a forget gate `f` , an output gate `o` , and a candidate cell variables $C^{\sim<t>}$
- Drawings (inspired by http://colah.github.io/posts/2015-08-Understanding-LSTMs/):



- Some variants on LSTM includes:
  - LSTM with **peephole connections**.
    - The normal LSTM with $C^{<t-1>}$ included with every gate.
- There isn't a universal superior between LSTM and it's variants. One of the advantages of GRU is that it's simpler and can be used to build much bigger network but the LSTM is more powerful and general.

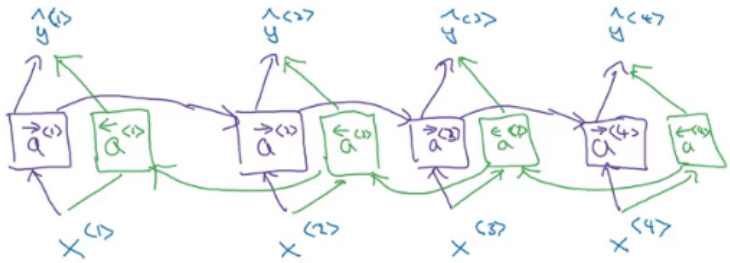
Bidirectional RNN

- There are still some ideas to let you build much more powerful sequence models. One of them is bidirectional RNNs and another is Deep RNNs.
- As we saw before, here is an example of the Name entity recognition task:



- The name **Teddy** cannot be learned from **He** and **said**, but can be learned from **bears**.
- BiRNNs fixes this issue.

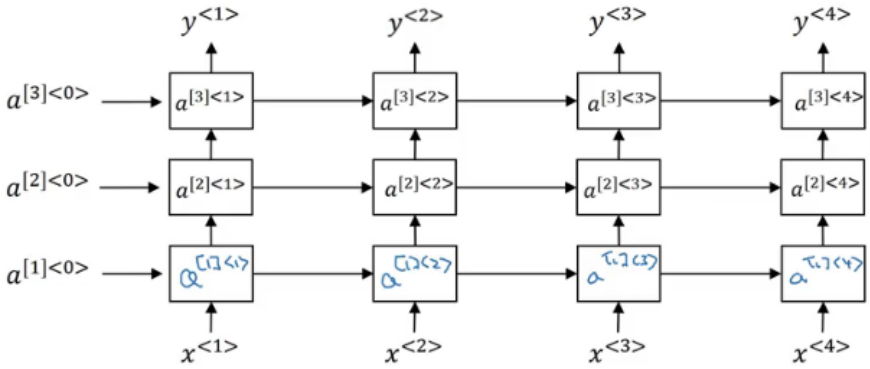
- Here is BRNNs architecture:



- Note, that BiRNN is an **acyclic graph**.
- Part of the forward propagation goes from left to right, and part - from right to left. It learns from both sides.
- To make predictions we use $\hat{y}^{<t>}$ by using the two activations that come from left and right.
- The blocks here can be any RNN block including the basic RNNs, LSTMs, or GRUs.
- For a lot of NLP or text processing problems, a BiRNN with LSTM appears to be commonly used.
- The disadvantage of BiRNNs that you need the entire sequence before you can process it. For example, in live speech recognition if you use BiRNNs you will need to wait for the person who speaks to stop to take the entire sequence and then make your predictions.

Deep RNNs

- In a lot of cases the standard one layer RNNs will solve your problem. But in some problems its useful to stack some RNN layers to make a deeper network.
- For example, a deep RNN with 3 layers would look like this:



- In feed-forward deep nets, there could be 100 or even 200 layers. In deep RNNs stacking 3 layers is already considered deep and expensive to train.
- In some cases you might see some feed-forward network layers connected after recurrent cell.

Back propagation with RNNs

- > In modern deep learning frameworks, you only have to implement the forward pass, and the framework takes care of the backward pass, so most deep learning engineers do not need to bother with the details of the backward pass. If however you are an expert in calculus and want to see the details of backprop in RNNs, you can work through this optional portion of the notebook.
- The quote is taken from this notebook. If you want the details of the back propagation with programming notes look at the linked notebook.

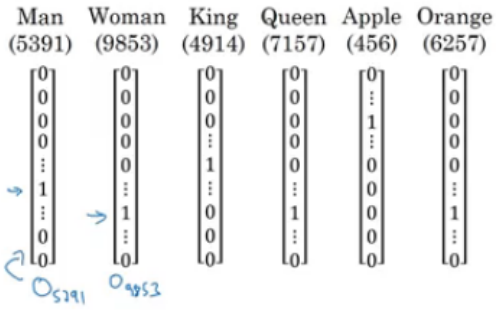
Natural Language Processing & Word Embeddings

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

Introduction to Word Embeddings

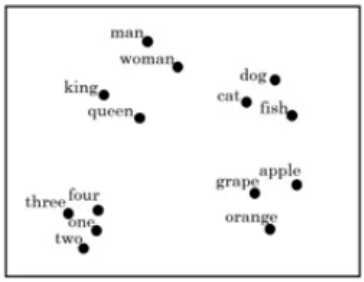
Word Representation

- NLP has been revolutionized by deep learning and especially by RNNs and deep RNNs.
- Word embeddings is a way of representing words. It lets your algorithm automatically understand the analogies between words like "king" and "queen".
- So far we have defined our language by a vocabulary. Then represented our words with a one-hot vector that represents the word in the vocabulary.
  - An image example would be:



  - We will use the annotation **O** $_{idx}$ for any word that is represented with one-hot like in the image.
  - One of the weaknesses of this representation is that it treats a word as a thing that itself and it doesn't allow an algorithm to generalize across words.
    - For example: "I want a glass of **orange** _____", a model should predict the next word as **juice**.
    - A similar example "I want a glass of **apple** _____", a model won't easily predict **juice** here if it wasn't trained on it. And if so the two examples aren't related although orange and apple are similar.
  - Inner product between any one-hot encoding vector is zero. Also, the distances between them are the same.
- So, instead of a one-hot presentation, won't it be nice if we can learn a featurized representation with each of these words: man, woman, king, queen, apple, and orange?

  - Each word will have a, for example, 300 features with a type of float point number.
  - Each word column will be a 300-dimensional vector which will be the representation.
  - We will use the notation $\mathbf{e}_{5391}$ to describe **man** word features vector.
  - Now, if we return to the examples we described again:
    - "I want a glass of **orange** _____"
    - I want a glass of **apple** _____
  - Orange and apple now share a lot of similar features which makes it easier for an algorithm to generalize between them.
  - We call this representation **Word embeddings**.
- To visualize word embeddings we use a t-SNE algorithm to reduce the features to 2 dimensions which makes it easy to visualize:

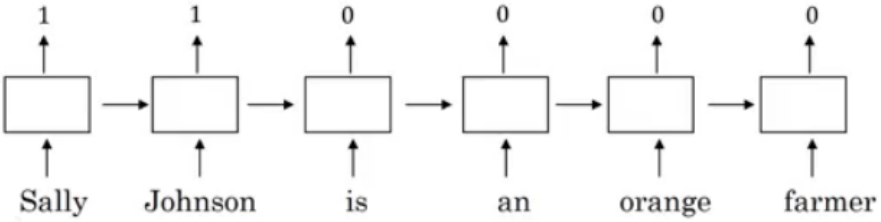


- You will get a sense that more related words are closer to each other.

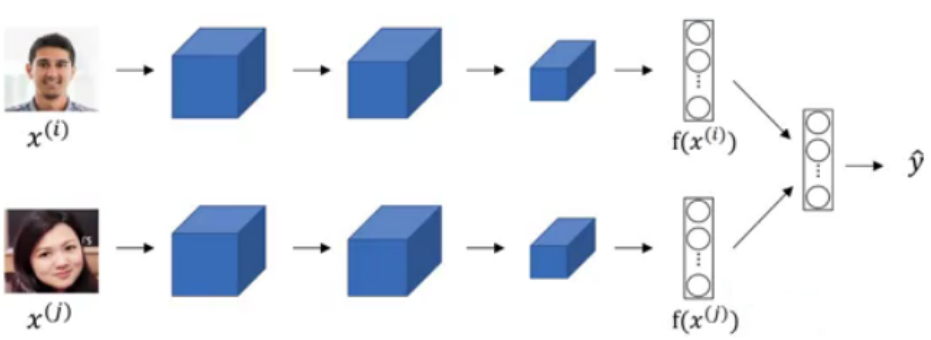
- The **word embeddings** came from that we need to embed a unique vector inside a n-dimensional space.

Using word embeddings

- Let's see how we can take the feature representation we have extracted from each word and apply it in the Named entity recognition problem.
- Given this example (from named entity recognition):



- **Sally Johnson** is a person's name.
- After training on this sentence the model should find out that the sentence "**Robert Lin** is an *apple* farmer" contains Robert Lin as a name, as apple and orange have near representations.
- Now if you have tested your model with this sentence "**Mahmoud Badry** is a *durian* cultivator" the network should learn the name even if it hasn't seen the word *durian* before (during training). That's the power of word representations.
- The algorithms that are used to learn **word embeddings** can examine billions of words of unlabeled text - for example, 100 billion words and learn the representation from them.
- Transfer learning and word embeddings:
  i. Learn word embeddings from large text corpus (1-100 billion of words).
    - Or download pre-trained embedding online.
  ii. Transfer embedding to new task with the smaller training set (say, 100k words).
  iii. Optional: continue to finetune the word embeddings with new data.
    - You bother doing this if your smaller training set (from step 2) is big enough.
- Word embeddings tend to make the biggest difference when the task you're trying to carry out has a relatively smaller training set.
- Also, one of the advantages of using word embeddings is that it reduces the size of the input!
  - 10,000 one hot compared to 300 features vector.
- Word embeddings have an interesting relationship to the face recognition task:



  - In this problem, we encode each face into a vector and then check how similar are these vectors.
  - Words **encoding** and **embeddings** have a similar meaning here.

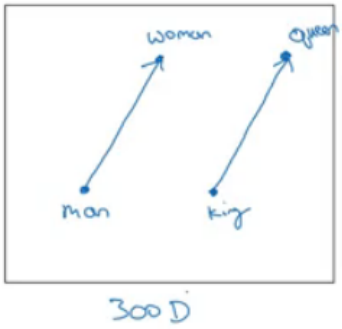
- In the word embeddings task, we are learning a representation for each word in our vocabulary (unlike in image encoding where we have to map each new image to some n-dimensional vector). We will discuss the algorithm in next sections.

Properties of word embeddings

- One of the most fascinating properties of word embeddings is that they can also help with analogy reasoning. While analogy reasoning may not be by itself the most important NLP application, but it might help convey a sense of what these word embeddings can do.
- Analogies example:
  - Given this word embeddings table:

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

  - Can we conclude this relation:
    - Man ==> Woman
    - King ==> ??
  - Lets subtract e_{Man} from e_{Woman}. This will equal the vector `[-2 0 0 0]`
  - Similar e_{King} - e_{Queen} = `[-2 0 0 0]`
  - So the difference is about the gender in both.



    - This vector represents the gender.
    - This drawing is a 2D visualization of the 4D vector that has been extracted by a t-SNE algorithm. It's a drawing just for visualization. Don't rely on the t-SNE algorithm for finding parallels.
  - So we can reformulate the problem to find:
    - e_{Man} - e_{Woman} ≈ e_{King} - $e_{??}$
  - It can also be represented mathematically by:

$$\arg\max_w \; sim(e_w, e_{king} - e_{man} + e_{woman})$$

  - It turns out that e_{Queen} is the best solution here that gets the the similar vector.
- Cosine similarity - the most commonly used similarity function:
  - Equation:

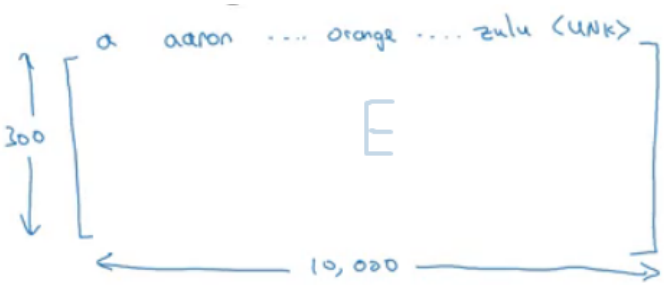
$$sim(u, v) = \frac{u^T v}{||u||_2 ||v||_2}$$

    - `CosineSimilarity(u, v) = u . v / ||u|| ||v||` = cos(θ)
    - The top part represents the inner product of `u` and `v` vectors. It will be large if the vectors are very similar.
- You can also use Euclidean distance as a similarity function (but it rather measures a dissimilarity, so you should take it with negative sign).

- We can use this equation to calculate the similarities between word embeddings and on the analogy problem where `u` = e_w and `v` = $e_{king} - e_{man} + e_{woman}$

Embedding matrix

- When you implement an algorithm to learn a word embedding, what you end up learning is a **embedding matrix**.
- Let's take an example:
  - Suppose we are using 10,000 words as our vocabulary (plus token).
  - The algorithm should create a matrix `E` of the shape (300, 10000) in case we are extracting 300 features.



  - If O_{6257} is the one hot encoding of the word **orange** of shape (10000, 1), then *np.dot(* `E` *,*O_{6257}*)* = e_{6257} which shape is (300, 1).
  - Generally *np.dot(* `E` *,* O_j*)* = e_j
- In the next sections, you will see that we first initialize `E` randomly and then try to learn all the parameters of this matrix.
- In practice it's not efficient to use a dot multiplication when you are trying to extract the embeddings of a specific word, instead, we will use slicing to slice a specific column. In Keras there is an embedding layer that extracts this column with no multiplication.

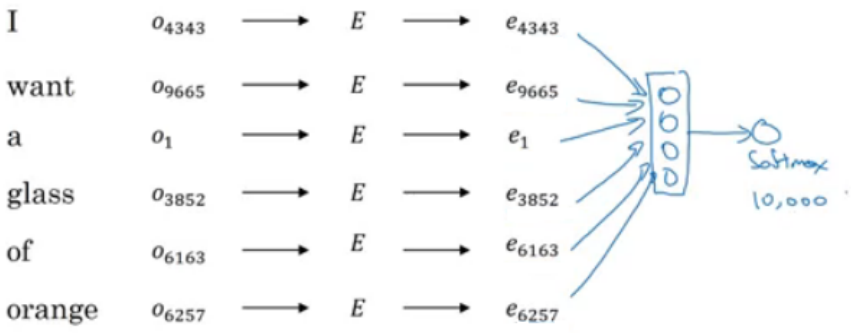
Learning Word Embeddings: Word2vec & GloVe

Learning word embeddings

- Let's start learning some algorithms that can learn word embeddings.
- At the start, word embeddings algorithms were complex but then they got simpler and simpler.
- We will start by learning the complex examples to make more intuition.
- **Neural language model**:
  - Let's start with an example:

I	want	a	glass	of	orange	_____.
4343	9665	1	3852	6163	6257	

  - We want to build a language model so that we can predict the next word.
  - So we use this neural network to learn the language model



    - We get e_j by `np.dot( E ,o<sub>j</sub>)`
    - NN layer has parameters `W1` and `b1` while softmax layer has parameters `W2` and `b2`
    - Input dimension is (300*6, 1) if the window size is 6 (six previous words).
    - Here we are optimizing `E` matrix and layers parameters. We need to maximize the likelihood to predict the next word given the context (previous words).
  - This model was build in 2003 and tends to work pretty decent for learning word embeddings.

- In the last example we took a window of 6 words that fall behind the word that we want to predict. There are other choices when we are trying to learn word embeddings.
  - Suppose we have an example: "I want a glass of orange **juice** to go along with my cereal"
  - To learn **juice**, choices of **context** are:
    a. Last 4 words.
      - We use a window of last 4 words (4 is a hyperparameter), "a glass of orange" and try to predict the next word from it.
    b. 4 words on the left and on the right.
      - "a glass of orange" and "to go along with"
    c. Last 1 word.
      - "orange"
    d. Nearby 1 word.
      - "glass" word is near juice.
      - This is the idea of **skip grams** model.
      - The idea is much simpler and works remarkably well.
      - We will talk about this in the next section.
- Researchers found that if you really want to build a *language model*, it's natural to use the last few words as a context. But if your main goal is really to learn a *word embedding*, then you can use all of these other contexts and they will result in very meaningful work embeddings as well.
- To summarize, the language modeling problem poses a machines learning problem where you input the context (like the last four words) and predict some target words. And posing that problem allows you to learn good word embeddings.

Word2Vec

- Before presenting Word2Vec, lets talk about **skip-grams**:
  - For example, we have the sentence: "I want a glass of orange juice to go along with my cereal"
  - We will choose **context** and **target**.
  - The target is chosen randomly based on a window with a specific size.

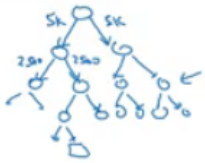
Context	Target	How far
orange	juice	+1
orange	glass	-2
orange	my	+6

    We have converted the problem into a supervised problem.
  - This is not an easy learning problem because learning within -10/+10 words (10 - an example) is hard.
  - We want to learn this to get our word embeddings model.
- Word2Vec model:
  - Vocabulary size = 10,000 words
  - Let's say that the context word are `c` and the target word is `t`
  - We want to learn `c` to `t`
  - We get e_c by `E . o_c`
  - We then use a softmax layer to get `P(t|c)` which is $\hat{y}$
  - Also we will use the cross-entropy loss function.
  - This model is called skip-grams model.

- The last model has a problem with the softmax layer:

$$p(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10,000} e^{\theta_j^T e_c}}$$

  - Here we are summing 10,000 numbers which corresponds to the number of words in our vocabulary.
  - If this number is larger say 1 million, the computation will become very slow.
- One of the solutions for the last problem is to use "**Hierarchical softmax classifier**" which works as a tree classifier.



- In practice, the hierarchical softmax classifier doesn't use a balanced tree like the drawn one. Common words are at the top and less common are at the bottom.
- How to sample the context **c**?
  - One way is to choose the context by random from your corpus.
  - If you have done it that way, there will be frequent words like "the, of, a, and, to, .." that can dominate other words like "orange, apple, durian,..."
  - In practice, we don't take the context uniformly random, instead there are some heuristics to balance the common words and the non-common words.
- word2vec paper includes 2 ideas of learning word embeddings. One is skip-gram model and another is CBoW (continuous bag-of-words).

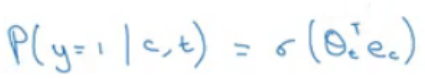
Negative Sampling

- Negative sampling allows you to do something similar to the skip-gram model, but with a much more efficient learning algorithm. We will create a different learning problem.
- Given this example:
  - "I want a glass of orange juice to go along with my cereal"
- The sampling will look like this:

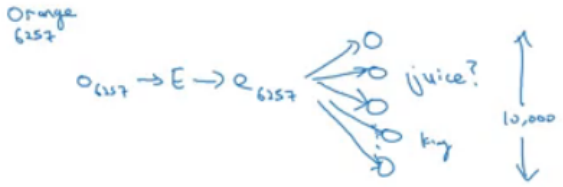
Context	Word	target
orange	juice	1
orange	king	0
orange	book	0
orange	the	0
orange	of	0

  We get positive example by using the same skip-grams technique, with a fixed window that goes around.
- To generate a negative example, we pick a word randomly from the vocabulary.
- Notice, that we got word "of" as a negative example although it appeared in the same sentence.
- So the steps to generate the samples are:
  i. Pick a positive context
  ii. Pick a k negative contexts from the dictionary.
- k is recommended to be from 5 to 20 in small datasets. For larger ones - 2 to 5.
- We will have a ratio of k negative examples to 1 positive ones in the data we are collecting.

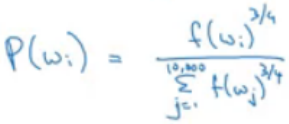
- Now let's define the model that will learn this supervised learning problem:
  - Lets say that the context word are `c` and the word are `t` and `y` is the target.
  - We will apply the simple logistic regression model.



  - The logistic regression model can be drawn like this:

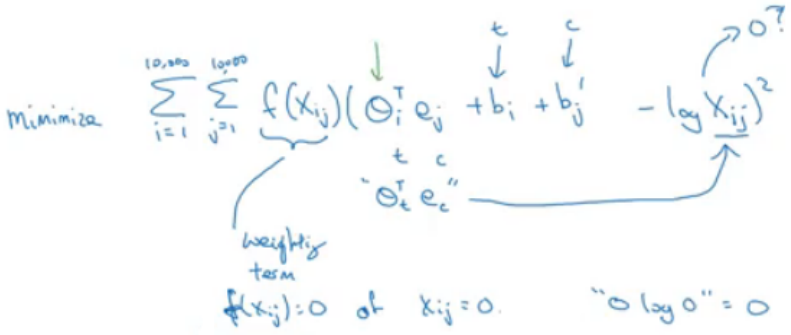


  - So we are like having 10,000 binary classification problems, and we only train k+1 classifier of them in each iteration.
- How to select negative samples:
  - We can sample according to empirical frequencies in words corpus which means according to how often different words appears. But the problem with that is that we will have more frequent words like *the, of, and...*
  - The best is to sample with this equation (according to authors):



GloVe word vectors

- GloVe is another algorithm for learning the word embedding. It's the simplest of them.
- This is not used as much as word2vec or skip-gram models, but it has some enthusiasts because of its simplicity.
- GloVe stands for Global vectors for word representation.
- Let's use our previous example: "I want a glass of orange juice to go along with my cereal".
- We will choose a context and a target from the choices we have mentioned in the previous sections.
- Then we will calculate this for every pair: X_{ct} = # times `t` appears in context of `c`
- $X_{ct} = X_{tc}$ if we choose a window pair, but they will not equal if we choose the previous words for example. In GloVe they use a window which means they are equal
- The model is defined like this:



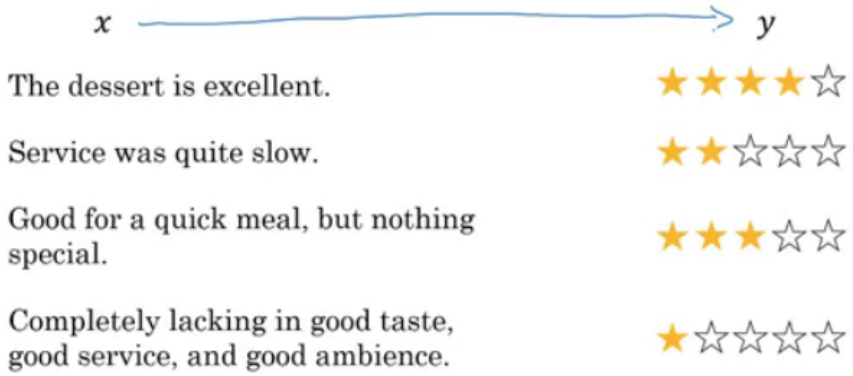
- f(x) - the weighting term, used for many reasons which include:
  - The `log(0)` problem, which might occur if there are no pairs for the given target and context values.
  - Giving not too much weight for stop words like "is", "the", and "this" which occur many times.
  - Giving not too little weight for infrequent words.
- **Theta** and **e** are symmetric which helps getting the final word embedding.

- *Conclusions on word embeddings:*
  - If this is your first try, you should try to download a pre-trained model that has been made and actually works best.
  - If you have enough data, you can try to implement one of the available algorithms.
  - Because word embeddings are very computationally expensive to train, most ML practitioners will load a pre-trained set of embeddings.
  - A final note that you can't guarantee that the axis used to represent the features will be well-aligned with what might be easily humanly interpretable axis like gender, royal, age.

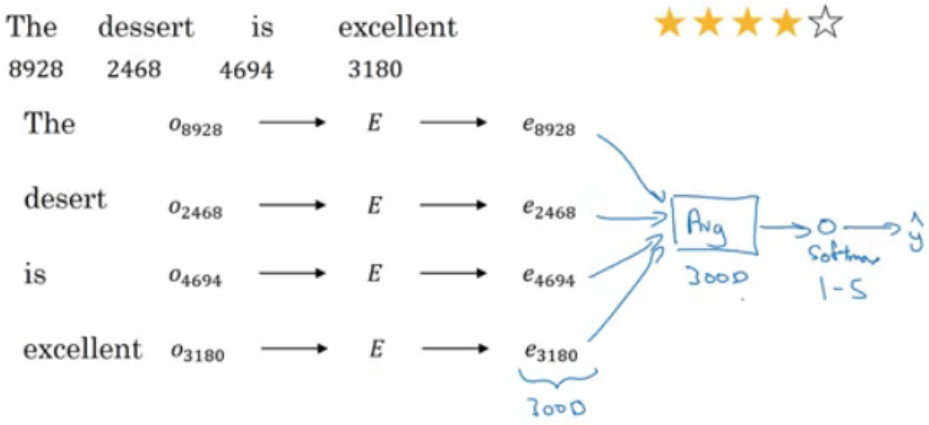
Applications using Word Embeddings

Sentiment Classification

- As we have discussed before, Sentiment classification is the process of finding if a text has a positive or a negative review. Its so useful in NLP and is used in so many applications. An example would be:

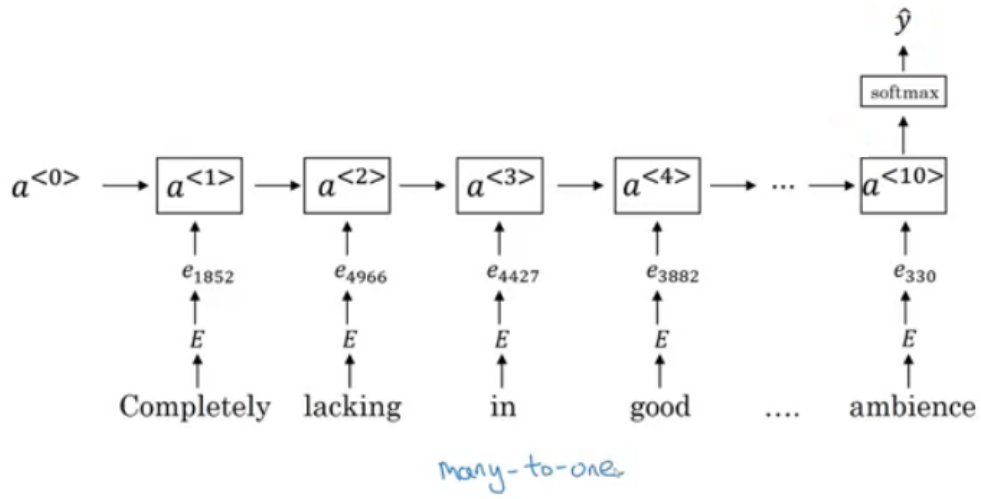


- One of the challenges with it, is that you might not have a huge labeled training data for it, but using word embeddings can help getting rid of this.
- The common dataset sizes varies from 10,000 to 100,000 words.
- A simple sentiment classification model would be like this:



  - The embedding matrix may have been trained on say 100 billion words.
  - Number of features in word embedding is 300.
  - We can use **sum** or **average** given all the words then pass it to a softmax classifier. That makes this classifier works for short or long sentences.
- One of the problems with this simple model is that it ignores words order. For example "Completely lacking in **good** taste, **good** service, and **good** ambience" has the word *good* 3 times but its a negative review.

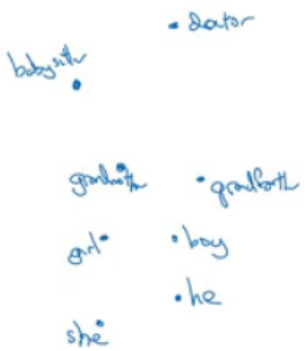
- A better model uses an RNN for solving this problem:



  - And so if you train this algorithm, you end up with a pretty decent sentiment classification algorithm.
  - Also, it will generalize better even if words weren't in your dataset. For example you have the sentence "Completely **absent** of good taste, good service, and good ambience", then even if the word "absent" is not in your label training set, if it was in your 1 billion or 100 billion word corpus used to train the word embeddings, it might still get this right and generalize much better even to words that were in the training set used to train the word embeddings but not necessarily in the label training set that you had for specifically the sentiment classification problem.

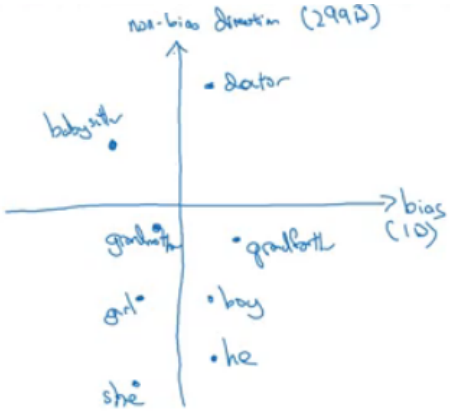
Debiasing word embeddings

- We want to make sure that our word embeddings are free from undesirable forms of bias, such as gender bias, ethnicity bias and so on.
- Horrifying results on the trained word embeddings in the context of Analogies:
  - Man : Computer_programmer as Woman : **Homemaker**
  - Father : Doctor as Mother : **Nurse**
- Word embeddings can reflect gender, ethnicity, age, sexual orientation, and other biases of text used to train the model.
- Learning algorithms by general are making important decisions and it mustn't be biased.
- Andrew thinks we actually have better ideas for quickly reducing the bias in AI than for quickly reducing the bias in the human race, although it still needs a lot of work to be done.
- Addressing bias in word embeddings steps:
  - Idea from the paper: https://arxiv.org/abs/1607.06520
  - Given these learned embeddings:



  - We need to solve the **gender bias** here. The steps we will discuss can help solve any bias problem but we are focusing here on gender bias.
  - Here are the steps:
    a. Identify the direction:
      - Calculate the difference between:
        - $e_{he} - e_{she}$
        - $e_{male} - e_{female}$
        - ....

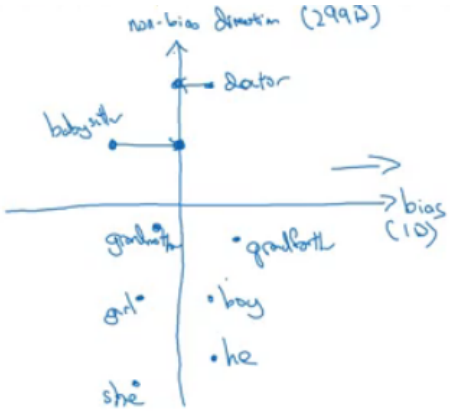
- Choose some k differences and average them.
- This will help you find this:



- By that we have found the bias direction which is 1D vector and the non-bias vector which is 299D vector.

b. Neutralize: For every word that is not definitional, project to get rid of bias.

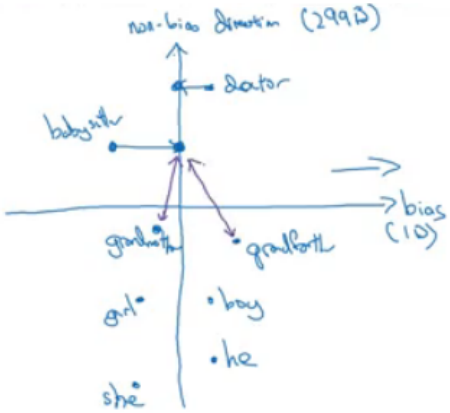
- Babysitter and doctor need to be neutral so we project them on non-bias axis with the direction of the bias:



  - After that they will be equal in the term of gender. - To do this the authors of the paper trained a classifier to tell the words that need to be neutralized or not.

c. Equalize pairs

- We want each pair to have difference only in gender. Like:
  - Grandfather - Grandmother - He - She - Boy - Girl
- We want to do this because the distance between grandfather and babysitter is bigger than babysitter and grandmother:



- To do that, we move grandfather and grandmother to a point where they will be in the middle of the non-bias axis.
- There are some words you need to do this for in your steps. Number of these words is relatively small.

Sequence models & Attention mechanism

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

Various sequence to sequence architectures

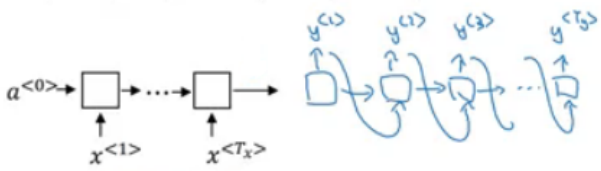
Basic Models

- In this section we will learn about sequence to sequence - *Many to Many* - models which are useful in various applications including machine translation and speech recognition.
- Let's start with the basic model:
  - Given this machine translation problem in which X is a French sequence and Y is an English sequence.

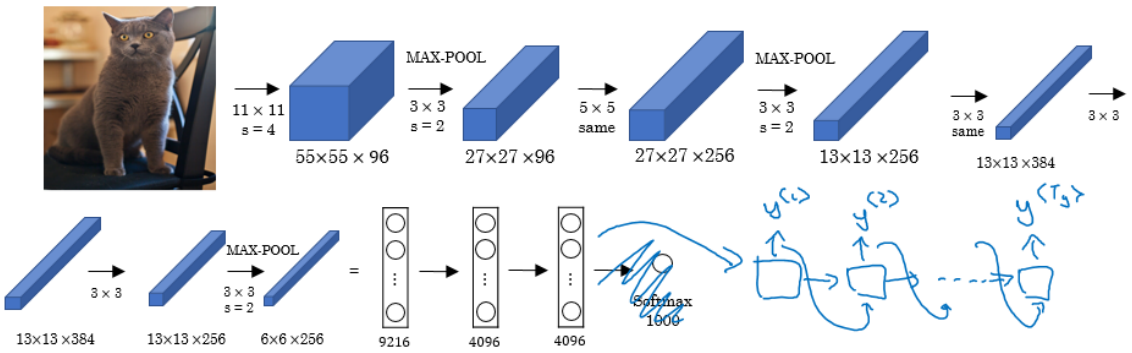
$$\underset{\text{Jane visite l'Afrique en septembre}}{x^{\langle 1\rangle}\ x^{\langle 2\rangle}\ x^{\langle 3\rangle}\ x^{\langle 4\rangle}\ x^{\langle 5\rangle}}$$

$$\longrightarrow \underset{y^{\langle 1\rangle}\ y^{\langle 2\rangle}\ y^{\langle 3\rangle}\ y^{\langle 4\rangle}\ y^{\langle 5\rangle}\ y^{\langle 6\rangle}}{\text{Jane is visiting Africa in September.}}$$

  - Our architecture will include **encoder** and **decoder**.
  - The encoder is RNN - LSTM or GRU are included - and takes the input sequence and then outputs a vector that should represent the whole input.
  - After that the decoder network, also RNN, takes the sequence built by the encoder and outputs the new sequence.



  - These ideas are from the following papers:
    - Sutskever et al., 2014. Sequence to sequence learning with neural networks
    - Cho et al., 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation
- An architecture similar to the mentioned above works for image captioning problem:
  - In this problem X is an image, while Y is a sentence (caption).
  - The model architecture image:

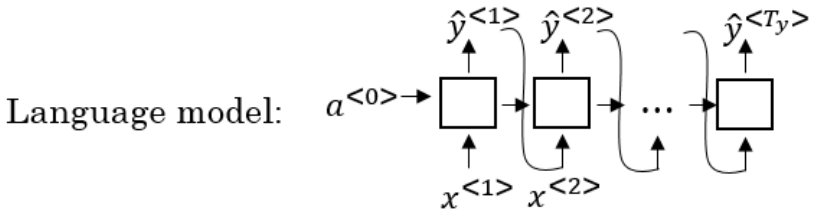


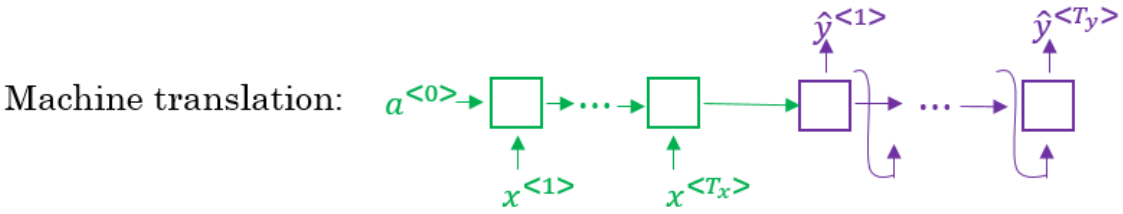
  - The architecture uses a pretrained CNN (like AlexNet) as an encoder for the image, and the decoder is an RNN.
  - Ideas are from the following papers (they share similar ideas):
    - Maoet et. al., 2014. Deep captioning with multimodal recurrent neural networks
    - Vinyals et. al., 2014. Show and tell: Neural image caption generator
    - Karpathy and Li, 2015. Deep visual-semantic alignments for generating image descriptions

Picking the most likely sentence

- There are some similarities between the language model we have learned previously, and the machine translation model we have just discussed, but there are some differences as well.

- The language model we have learned is very similar to the decoder part of the machine translation model, except for $a^{<0>}$





- Problems formulations also are different:
  - In language model: P($y^{<1>}$, ..., $y^{<Ty>}$)
  - In machine translation: P($y^{<1>}$, ..., $y^{<Ty>}$ | $x^{<1>}$, ..., $x^{<Tx>}$)
- What we don't want in machine translation model, is not to sample the output at random. This may provide some choices as an output. Sometimes you may sample a bad output.
  - Example:
    - X = "Jane visite l'Afrique en septembre."
    - Y may be:
      - Jane is visiting Africa in September.
      - Jane is going to be visiting Africa in September.
      - In September, Jane will visit Africa.
- So we need to get the best output it can be:

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

- The most common algorithm is the beam search, which we will explain in the next section.
- Why not use greedy search? Why not get the best choices each time?
  - It turns out that this approach doesn't really work!
  - Lets explain it with an example:
    - The best output for the example we talked about is "Jane is visiting Africa in September."
    - Suppose that when you are choosing with greedy approach, the first two words were "Jane is", the word that may come after that will be "going" as "going" is the most common word that comes after " is" so the result may look like this: "Jane is going to be visiting Africa in September.". And that isn't the best/optimal solution.
- So what is better than greedy approach, is to get an approximate solution, that will try to maximize the output (the last equation above).

Beam Search

- Beam search is the most widely used algorithm to get the best output sequence. It's a heuristic search algorithm.
- To illustrate the algorithm we will stick with the example from the previous section. We need Y = "Jane is visiting Africa in September."
- The algorithm has a parameter `B` which is the beam width. Lets take `B = 3` which means the algorithm will get 3 outputs at a time.
- For the first step you will get ["in", "jane", "september"] words that are the best candidates.
- Then for each word in the first output, get B next (second) words and select top best B combinations where the best are those what give the highest value of multiplying both probabilities - P($y^{<1>}$|x) * P($y^{<2>}$|x,$y^{<1>}$). Se we will have then ["in september", "jane is", "jane visit"]. Notice, that we automatically discard *september* as a first word.
- Repeat the same process and get the best B words for ["september", "is", "visit"] and so on.
- In this algorithm, keep only B instances of your network.
- If `B = 1` this will become the greedy search.

Refinements to Beam Search

- In the previous section, we have discussed the basic beam search. In this section, we will try to do some refinements to it.
- The first thing is **Length optimization**
  - In beam search we are trying to optimize:

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

  - And to do that we multiply:
    $P(y^{<1>} | x) * P(y^{<2>} | x, y^{<1>}) * ... * P(y^{<t>} | x, y^{<y(t-1)>})$
  - Each probability is a fraction, most of the time a small fraction.
  - Multiplying small fractions will cause a **numerical overflow**. Meaning that it's too small for the floating part representation in your computer to store accurately.
  - So in practice we use **summing logs of probabilities** instead of multiplying directly.

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

  - But there's another problem. The two optimization functions we have mentioned are preferring small sequences rather than long ones. Because multiplying more fractions gives a smaller value, so fewer fractions - bigger result.
  - So there's another step - dividing by the number of elements in the sequence.

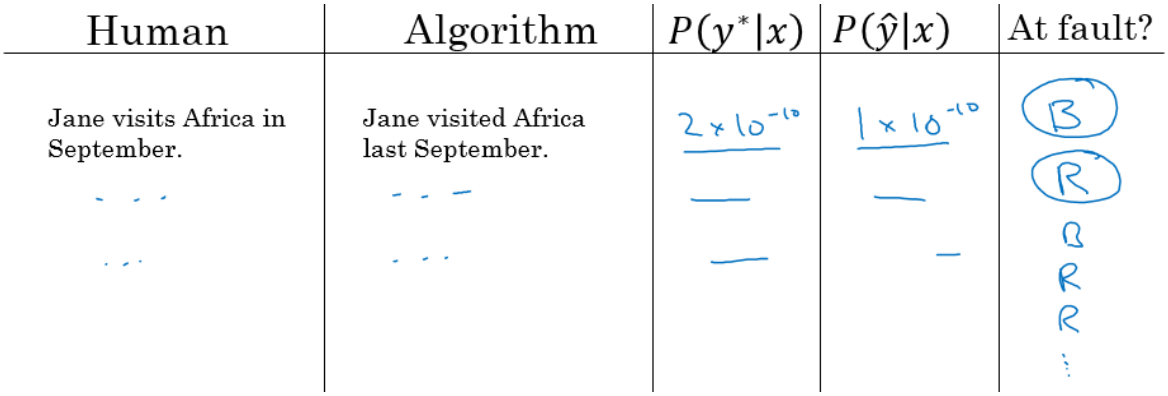
$$\frac{1}{T_y^{\alpha}}\sum_{t=1}^{T_y}\log P(y^{<t>}|x,y^{<1>},\dots,y^{<t-1>})$$

    - alpha is a hyperparameter to tune.
    - If alpha = 0 - no sequence length normalization.
    - If alpha = 1 - full sequence length normalization.
    - In practice alpha = 0.7 is a good thing (somewhere in between two extremes).
- The second thing is how can we choose best `B` ?
  - The larger B - the larger possibilities, the better are the results. But it will be more computationally expensive.
  - In practice, you might see in the production setting `B=10`
  - `B=100`, `B=1000` are uncommon (sometimes used in research settings)
  - Unlike exact search algorithms like BFS (Breadth First Search) or DFS (Depth First Search), Beam Search runs faster but is not guaranteed to find the exact solution.

Error analysis in beam search

- We have talked before on **Error analysis** in *"Structuring Machine Learning Projects"* course. We will apply these concepts to improve our beam search algorithm.
- We will use error analysis to figure out if the `B` hyperparameter of the beam search is the problem (it doesn't get an optimal solution) or in our RNN part.
- Let's take an example:
  - Initial info:
    - x = "Jane visite l'Afrique en septembre."
    - y^* = "Jane visits Africa in September." - right answer
    - $\hat{y}$ = "Jane visited Africa last September." - answer produced by model
  - Our model that has produced not a good result.
  - We now want to know who to blame - the RNN or the beam search.
  - To do that, we calculate $P(y^* | X)$ and $P(\hat{y} | X)$. There are two cases:
    - Case 1 ($P(y^* | X) > P(\hat{y} | X)$):
      - Conclusion: Beam search is at fault.
    - Case 2 ($P(y^* | X) <= P(\hat{y} | X)$):
      - Conclusion: RNN model is at fault.
- The error analysis process is as following:

- You choose N error examples and make the following table:



  - B for beam search, R is for the RNN.
  - Get counts and decide what to work on next.

BLEU Score

- One of the challenges of machine translation, is that given a sentence in a language there are one or more possible good translation in another language. So how do we evaluate our results?
- The way we do this is by using **BLEU score**. BLEU stands for *bilingual evaluation understudy*.
- The intuition is: as long as the machine-generated translation is pretty close to any of the references provided by humans, then it will get a high BLEU score.
- Let's take an example:
  - X = "Le chat est sur le tapis."
  - Y1 = "The cat is on the mat." (human reference 1)
  - Y2 = "There is a cat on the mat." (human reference 2)
  - Suppose that the machine outputs: "the the the the the the the."
  - One way to evaluate the machine output is to look at each word in the output and check if it is in the references. This is called *precision*:
    - precision = 7/7 because "the" appeared in Y1 or Y2
  - This is not a useful measure!
  - We can use a modified precision in which we are looking for the reference with the maximum number of a particular word and set the maximum appearing of this word to this number. So:
    - modified precision = 2/7 because the max is 2 in Y1
    - We clipped the 7 times by the max which is 2.
  - Here we are looking at one word at a time - unigrams, we may look at n-grams too
- BLEU score on bigrams
  - The **n-grams** typically are collected from a text or speech corpus. When the items are words, **n-grams** may also be called shingles. An **n-gram** of size 1 is referred to as a "unigram"; size 2 is a "bigram" (or, less commonly, a "digram"); size 3 is a "trigram".
  - X = "Le chat est sur le tapis."
  - Y1 = "The cat is on the mat."
  - Y2 = "There is a cat on the mat."
  - Suppose that the machine outputs: "the cat the cat on the mat."
  - The bigrams in the machine output:

Pairs	Count	Count clip
the cat	2	1 (Y1)
cat the	1	0
cat on	1	1 (Y2)
on the	1	1 (Y1)

Pairs	Count	Count clip
the mat	1	1 (Y1)
Totals	6	4

Modified precision = sum(Count clip) / sum(Count) = 4/6

- So here are the equations for modified precision for the n-grams case:

$$p_1 = \frac{\sum_{unigram \in \hat{y}} count_{clip}\,(unigram)}{\sum_{unigram \in \hat{y}} count\,(unigram)} \qquad p_n = \frac{\sum_{ngram \in \hat{y}} count_{clip}\,(ngram)}{\sum_{ngram \in \hat{y}} count\,(ngram)}$$

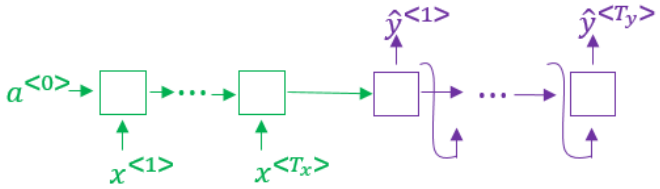
- Let's put this together to formalize the BLEU score:
  - **P_n** = Bleu score on one type of n-gram
  - **Combined BLEU score** = BP * exp(1/n * sum(P_n))
    - For example if we want BLEU for 4, we compute P_1, P_2, P_3, P_4 and then average them and take the exp.
  - **BP** is called **BP penalty** which stands for brevity penalty. It turns out that if a machine outputs a small number of words it will get a better score so we need to handle that.

$$BP = \begin{cases} 1 & \text{if MT\_output\_length} > \text{reference\_output\_length} \\ \exp(1 - \text{MT\_output\_length}/\text{reference\_output\_length}) & \text{otherwise} \end{cases}$$

- BLEU score has several open source implementations.
- It is used in a variety of systems like machine translation and image captioning.

Attention Model Intuition

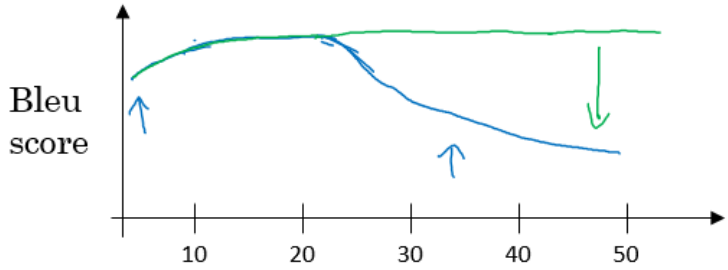
- So far we were using sequence to sequence models with an encoder and decoders. There is a technique called *attention* which makes these models even better.
- The attention idea has been one of the most influential ideas in deep learning.
- The problem of long sequences:
  - Given this model, inputs, and outputs.



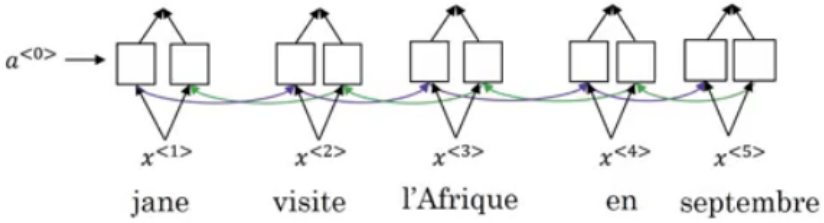
Jane s'est rendue en Afrique en septembre dernier, a apprécié la culture et a rencontré beaucoup de gens merveilleux; elle est revenue en parlant comment son voyage était merveilleux, et elle me tente d'y aller aussi.

Jane went to Africa last September, and enjoyed the culture and met many wonderful people; she came back raving about how wonderful her trip was, and is tempting me to go too.

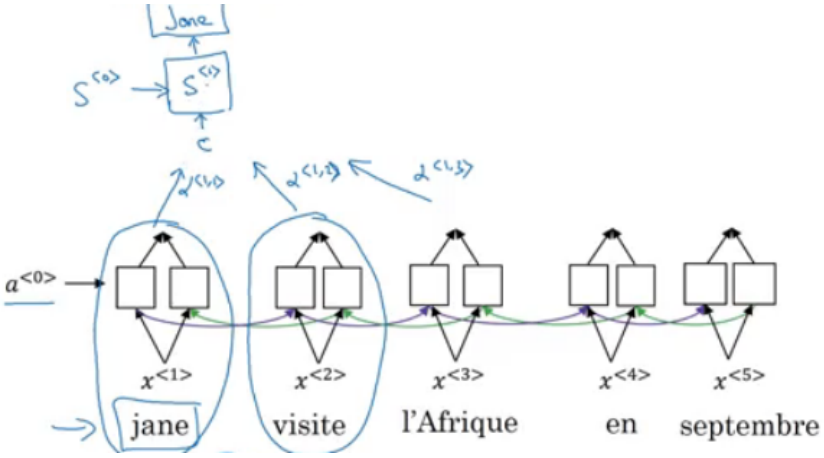
  - The encoder should memorize this long sequence into one vector, and the decoder has to process this vector to generate the translation.
  - If a human would translate this sentence, he/she wouldn't read the whole sentence and memorize it then try to translate it. He/she translates a part at a time.
  - The performance of this model decreases if a sentence is long.
  - We will discuss the attention model that works like a human that looks at parts at a time. That will significantly increase the accuracy even with longer sequence:



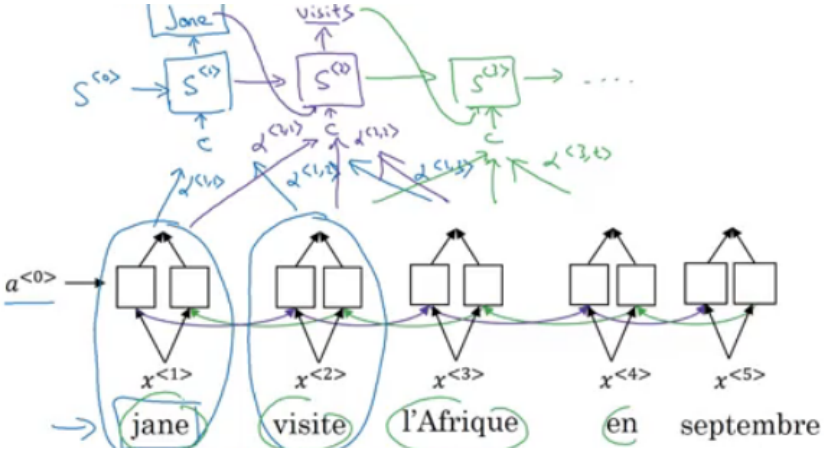
  - Blue is the normal model, while green is the model with attention mechanism.
- In this section we will give just some intuitions about the attention model and in the next section we will discuss it's details.
- At first the attention model was developed for machine translation but then other applications used it like computer vision and new architectures like Neural Turing machine.
- The attention model was descried in this paper:
  - Bahdanau et. al., 2014. Neural machine translation by jointly learning to align and translate
- Now for the intuition:
  - Suppose that our encoder is a bidirectional RNN:



  - We give the French sentence to the encoder and it should generate a vector that represents the inputs.
  - Now to generate the first word in English which is "Jane" we will make another RNN which is the decoder.
  - Attention weights are used to specify which words are needed when to generate a word. So to generate "jane" we will look at "jane", "visite", "l'Afrique"

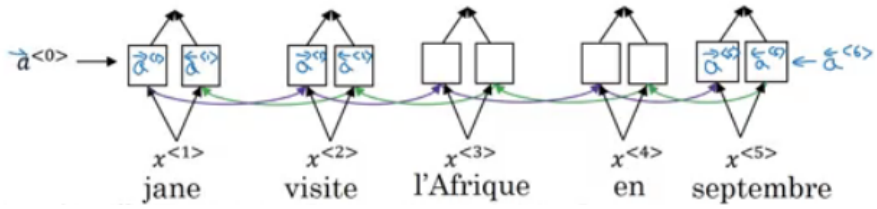


  - alpha$^{<1,1>}$, alpha$^{<1,2>}$, and alpha$^{<1,3>}$ are the attention weights being used.
  - And so to generate any word there will be a set of attention weights that controls which words we are looking at right now.

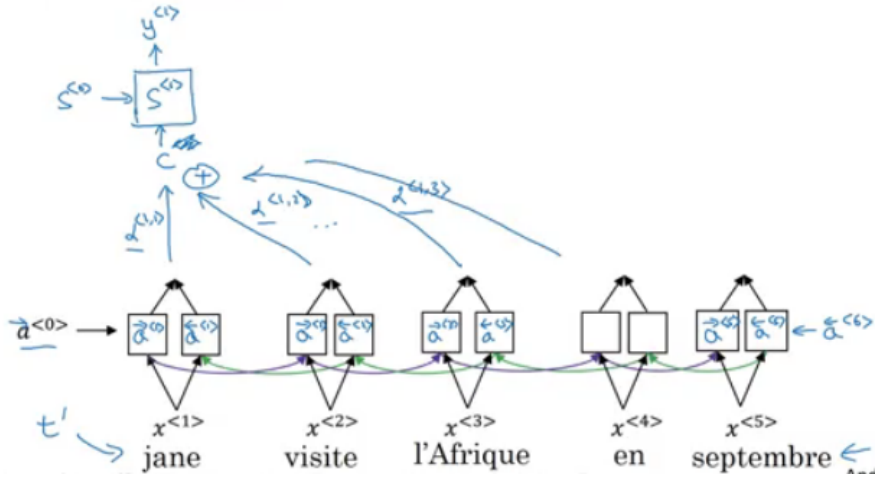


Attention Model

- Lets formalize the intuition from the last section into the exact details on how this can be implemented.
- First we will have an bidirectional RNN (most common is LSTMs) that encodes French language:



- For learning purposes, lets assume that $a^{<t'>}$ will include the both directions activations at time step t'.
- We will have a unidirectional RNN to produce the output using a context c which is computed using the attention weights, which denote how much information does the output needs to look in $a^{<t'>}$



- Sum of the attention weights for each element in the sequence should be 1:

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

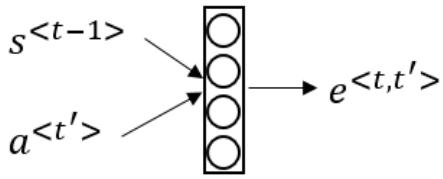
- The context c is calculated using this equation:

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

- Lets see how can we compute the attention weights:
  - So $alpha^{<t, t'>}$ = amount of attention $y^{<t>}$ should pay to $a^{<t'>}$
    - Like for example we payed attention to the first three words through $alpha^{<1,1>}$, $alpha^{<1,2>}$, $alpha^{<1,3>}$
  - We are going to softmax the attention weights so that their sum is 1:

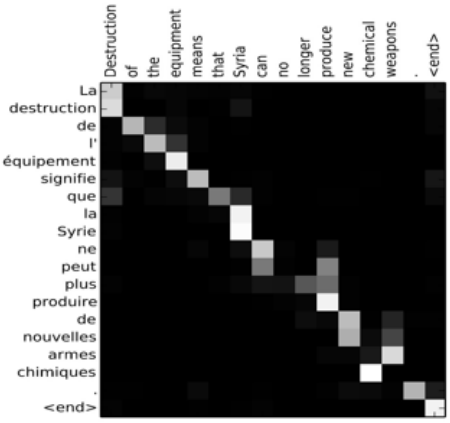
$$\alpha^{<t,t'>} = \frac{\exp(e^{<t,t'>})}{\sum_{t'=1}^{Tx} \exp(e^{<t,t'>})}$$

  - Now we need to know how to calculate $e^{<t, t'>}$. We will compute e using a small neural network (usually 1-layer, because we will need to compute this a lot):



    - $s^{<t-1>}$ is the hidden state of the RNN s, and $a^{<t'>}$ is the activation of the other bidirectional RNN.
- One of the disadvantages of this algorithm is that it takes quadratic time or quadratic cost to run.

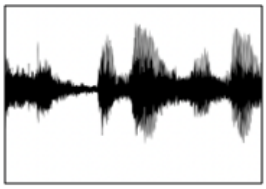
- One fun way to see how attention works is by visualizing the attention weights:



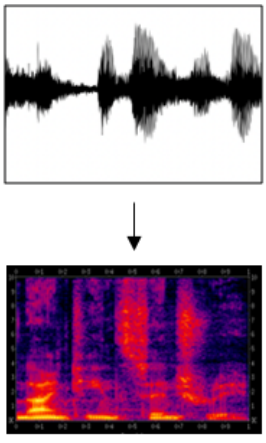
Speech recognition - Audio data

Speech recognition

- One of the most exciting developments using sequence-to-sequence models has been the rise of very accurate speech recognition.
- Let's define the speech recognition problem:
  - X: audio clip
  - Y: transcript
  - If you plot an audio clip it will look like this:



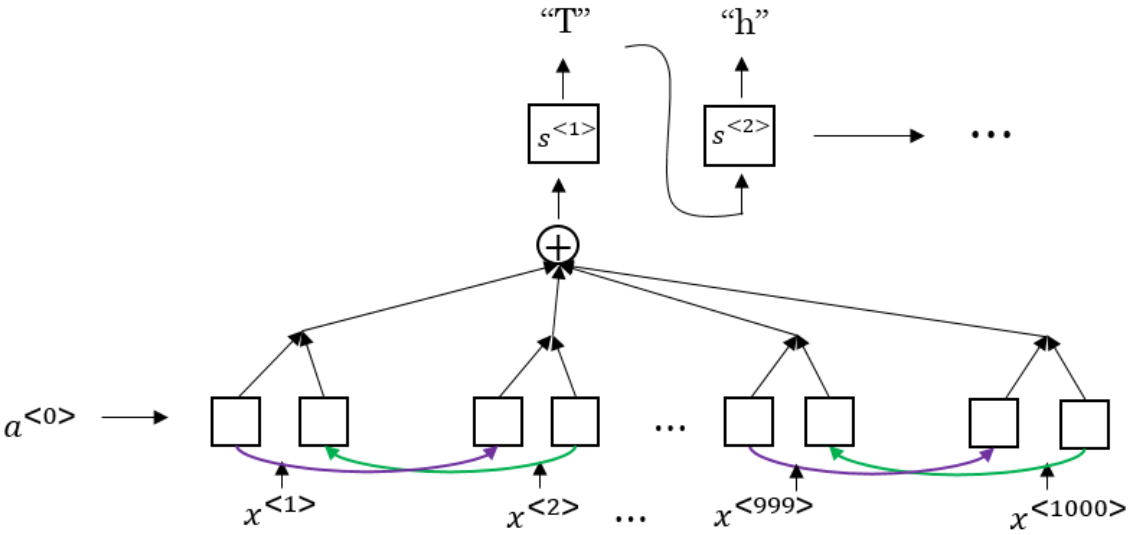
    - The horizontal axis is time while the vertical is changes in air pressure.
  - What really is an audio recording? A microphone records little variations in air pressure over time, and it is these little variations in air pressure that your ear perceives as sound. You can think of an audio recording is a long list of numbers measuring the little air pressure changes detected by the microphone. We will use audio sampled at 44100 Hz (or 44100 Hertz). This means the microphone gives us 44100 numbers per second. Thus, a 10 second audio clip is represented by 441000 numbers (= 10 * 44100).
  - It is quite difficult to work with "raw" representation of audio.
  - Because even human ear doesn't process raw wave forms, the human ear can process different frequencies.
  - There's a common preprocessing step for an audio - generate a spectrogram which works similarly to human ears.



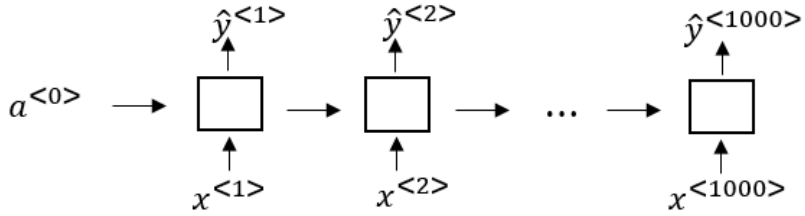
    - The horizontal axis is time while the vertical is frequencies. Intensity of different colors shows the amount of energy - how loud is the sound for different frequencies (a human ear does a very similar preprocessing step).
  - A spectrogram is computed by sliding a window over the raw audio signal, and calculates the most active frequencies in each window using a Fourier transformation.

- In the past days, speech recognition systems were built using *phonemes* that are a hand engineered basic units of sound. Linguists used to hypothesize that writing down audio in terms of these basic units of sound called *phonemes* would be the best way to do speech recognition.
- End-to-end deep learning found that phonemes was no longer needed. One of the things that made this possible is the large audio datasets.
- Research papers have around 300 - 3000 hours of training data while the best commercial systems are now trained on over 100,000 hours of audio.

- You can build an accurate speech recognition system using the attention model that we have descried in the previous section:



- One of the methods that seem to work well is *CTC cost* which stands for "Connectionist temporal classification"
  - To explain this let's say that Y = "the quick brown fox"
  - We are going to use an RNN with input, output structure:

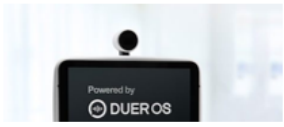


  - Note: this is a unidirectional RNN, but in practice a bidirectional RNN is used.
  - Notice, that the number of inputs and number of outputs are the same here, but in speech recognition problem input X tends to be a lot larger than output Y.
    - 10 seconds of audio at 100Hz gives us X with shape (1000,). These 10 seconds don't contain 1000 character outputs.
  - The CTC cost function allows the RNN to output something like this:
    - `ttt_h_eee<SPC>___<SPC>qqq___` - this covers "the q".
    - The _ is a special character called "blank" and `<SPC>` is for the "space" character.
    - Basic rule for CTC: collapse repeated characters not separated by "blank"
  - So the 19 character in our Y can be generated into 1000 character output using CTC and it's special blanks.
  - The ideas were taken from this paper:
    - Graves et al., 2006. Connectionist Temporal Classification: Labeling unsegmented sequence data with recurrent neural networks
    - This paper's ideas were also used by Baidu's DeepSpeech.
- Using both attention model and CTC cost can help you to build an accurate speech recognition system.

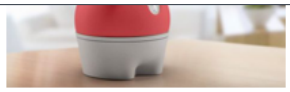
Trigger Word Detection

- With the rise of deep learning speech recognition, there are a lot of devices that can be waked up by saying some words with your voice. These systems are called trigger word detection systems.
- For example, Alexa - a smart device made by Amazon - can answer your call "Alexa, what time is it?" and then Alexa will respond to you.

- Trigger word detection systems include:

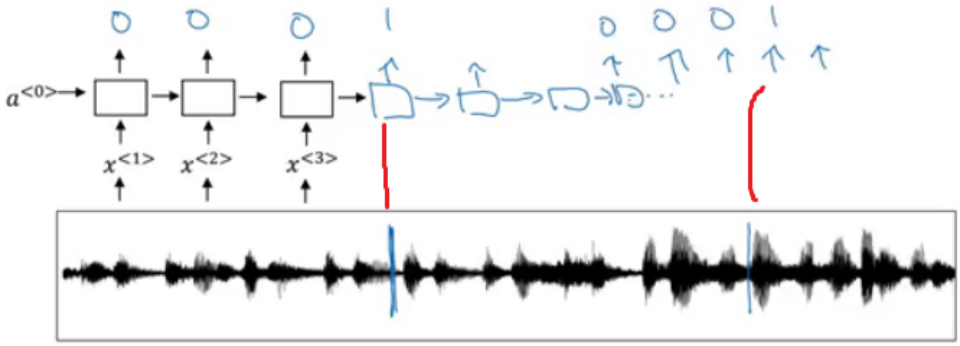




Readme.md

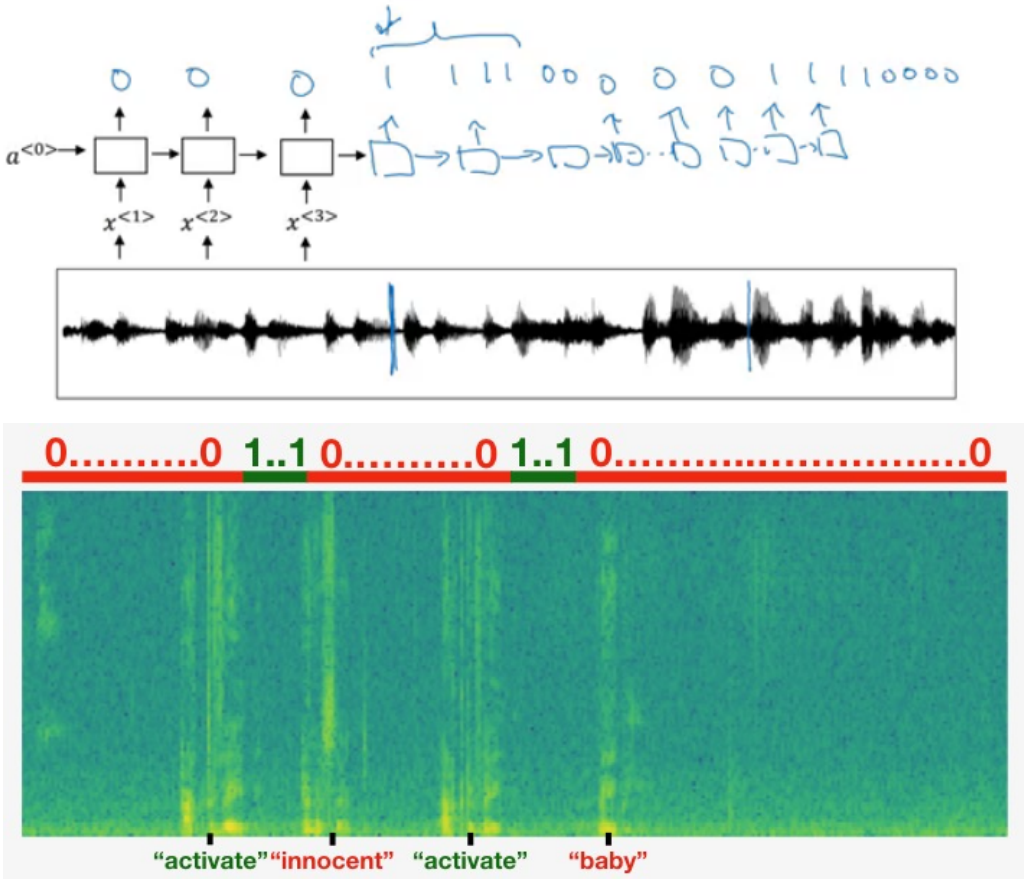


Amazon Echo (Alexa) | Baidu DuerOS (xiaodunihao) | Apple Siri (Hey Siri) | Google Home (Okay Google)

- For now, the trigger word detection literature is still evolving so there actually isn't a single universally agreed on the algorithm for trigger word detection yet. But let's discuss an algorithm that can be used.
- Let's now build a model that can solve this problem:
  - X: audio clip
  - X has been preprocessed and spectrogram features have been returned of X
    - $X^{<1>}, X^{<2>}, \ldots, X^{<t>}$
  - Y will be labels 0 or 1. 0 represents the non-trigger word, while 1 is that trigger word that we need to detect.
  - The model architecture can be like this:



    - The vertical lines in the audio clip represent moment just after the trigger word. The corresponding to this will be 1.
  - One disadvantage of this creates a very imbalanced training set. There will be a lot of zeros and few ones.
  - A hack to solve this is to make an output a few ones for several times or for a fixed period of time before reverting back to zero.

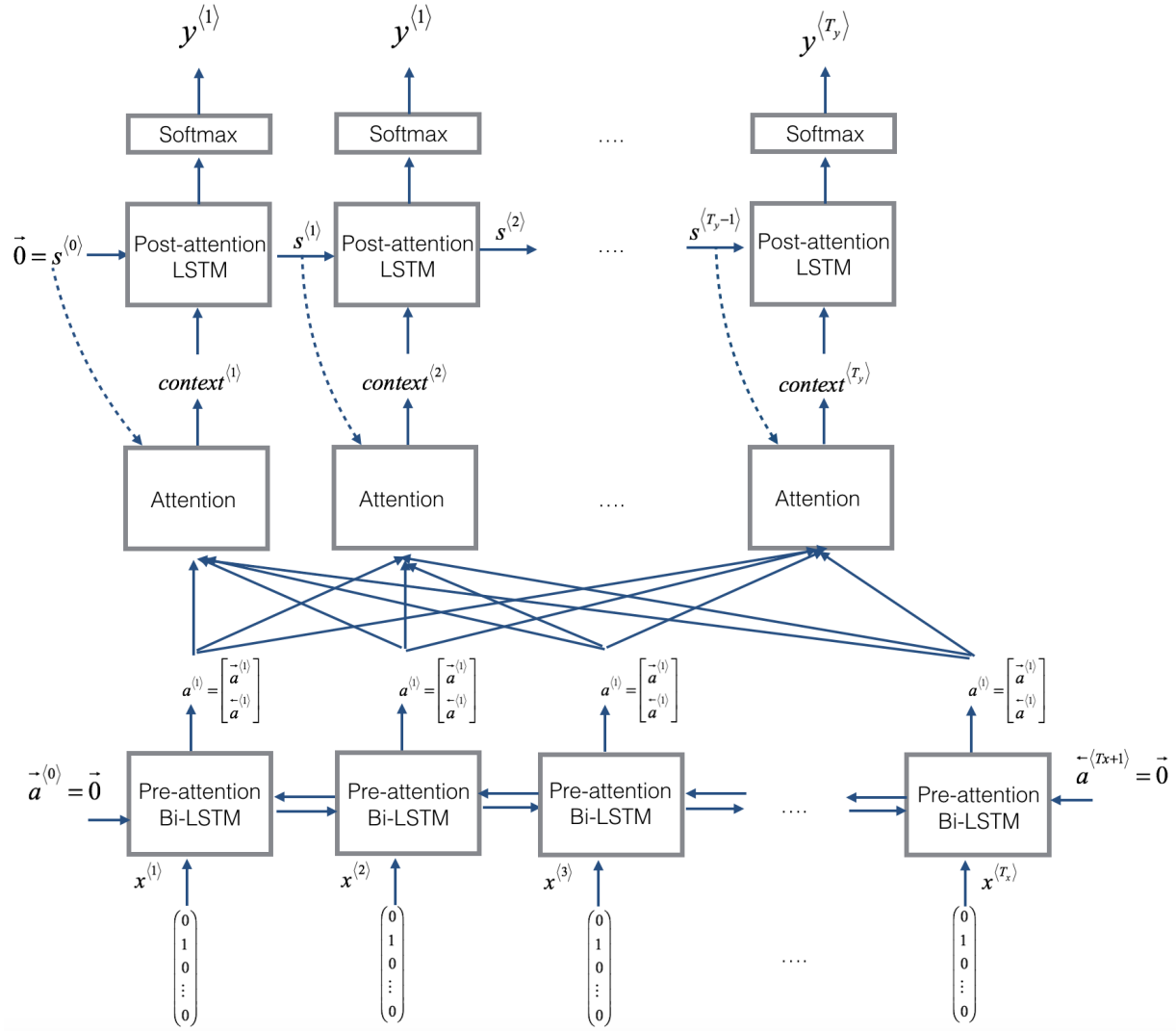


Extras

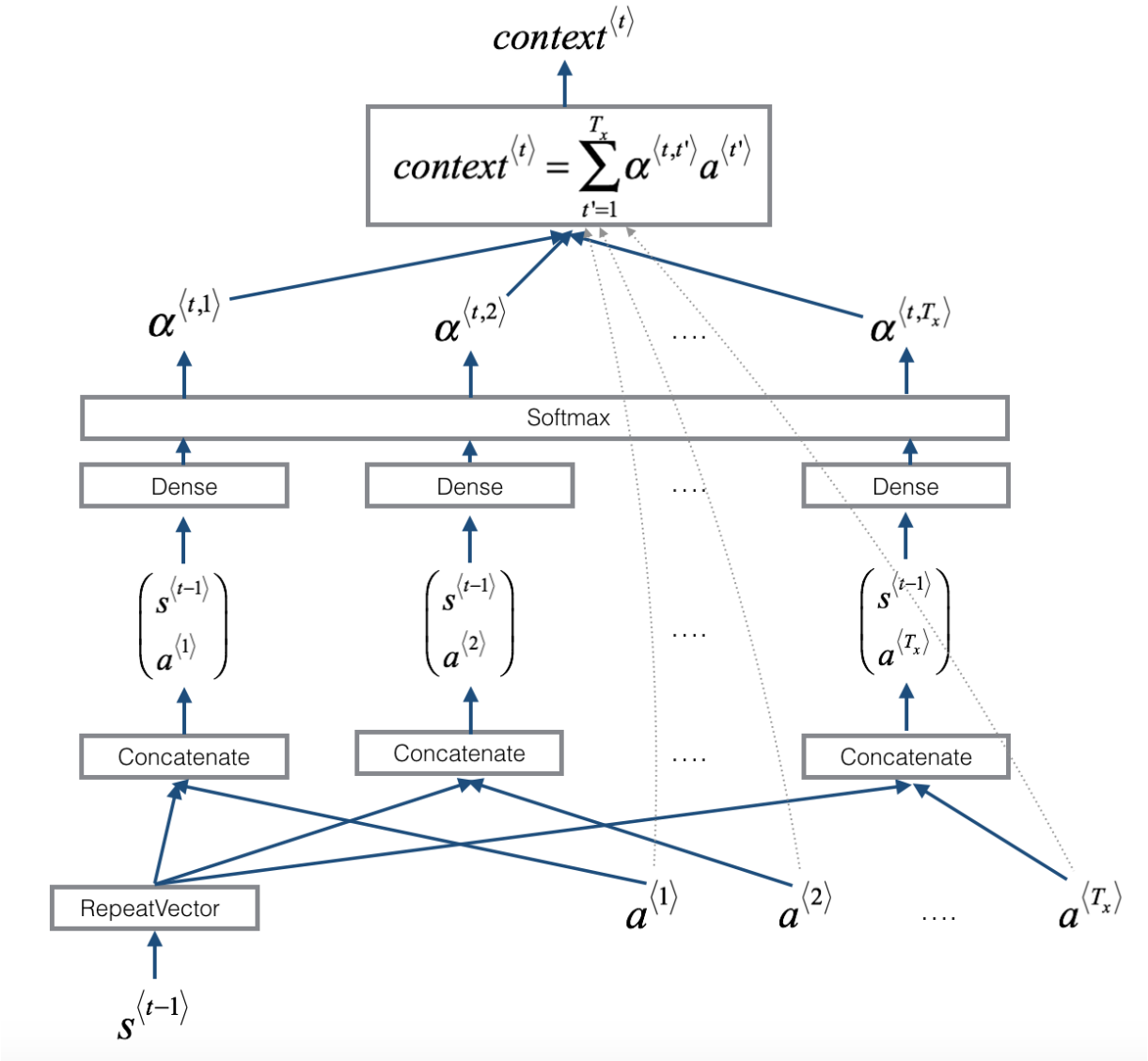
Machine translation attention model (from notebooks)

- The model is built with keras layers.

- The attention model.



  - There are two separate LSTMs in this model. Because the one at the bottom of the picture is a Bi-directional LSTM and comes *before* the attention mechanism, we will call it *pre-attention* Bi-LSTM. The LSTM at the top of the diagram comes *after* the attention mechanism, so we will call it the *post-attention* LSTM. The pre-attention Bi-LSTM goes through T_x time steps; the post-attention LSTM goes through T_y time steps.
  - The post-attention LSTM passes $s^{<t>}$, $c^{<t>}$ from one time step to the next. In the lecture videos, we were using only a basic RNN for the post-activation sequence model, so the state captured by the RNN output activations $s^{<t>}$. But since we are using an LSTM here, the LSTM has both the output activation $s^{<t>}$ and the hidden cell state $c^{<t>}$. However, unlike previous text generation examples (such as Dinosaurus in week 1), in this model the post-activation LSTM at time `t` does will not take the specific generated $y^{<t-1>}$ as input; it only takes $s^{<t>}$ and $c^{<t>}$ as input. We have designed the model this way, because (unlike language generation where adjacent characters are highly correlated) there isn't as strong a dependency between the previous character and the next character in a YYYY-MM-DD date.
- What one "Attention" step does to calculate the attention variables $\alpha^{<t,\ t>}$, which are used to compute the context variable $context^{<t>}$ for each timestep in the output (t=1, ..., T_y).



- The diagram uses a `RepeatVector` node to copy $s^{<t-1>}$'s value T_x times, and then `Concatenation` to concatenate $s^{<t-1>}$ and $a^{<t>}$ to compute $e^{<t,\ t>}$, which is then passed through a softmax to compute $\alpha^{<t,\ t>}$.

These Notes were made by Mahmoud Badry @2018