


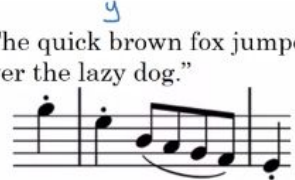


COURSE 5: SEQUENCE MODELS	2
Recurrent Neural Networks	2
Why sequence models	2
Notation	2
Recurrent Neural Network Model	4
Backpropagation through time	7
Different types of RNNs	8
Language model and sequence generation	10
Sampling novel sequences	12
Vanishing gradients with RNNs	13
Gated Recurrent Unit (GRU)	14
Long Short Term Memory (LSTM)	16
Bidirectional RNN	18
Deep RNNs	19
NLP & Word Embeddings - Introduction to Word Embeddings	19
Word Representation	19
Using Word Embeddings	22
Properties of word embeddings	23
Embedding matrix	24
NLP & Word Embeddings - Learning Word Embeddings: Word2vec & GloVe	26
Learning word embeddings	26
Word2Vec	27
Negative Sampling	28
GloVe word vectors	29
Sentiment Classification	30
Debiasing word embeddings	31
Sequence models & Attention mechanism - Sequence to sequence architectures	34
Basic Models	34
Picking the most likely sentence	36
Beam Search	37
Refinements to Beam Search	39
Error analysis in beam search	41
Attention Models	42
Speech Recognition	45
Trigger Word Detection	47

COURSE 5: SEQUENCE MODELS

Recurrent Neural Networks

Why sequence models

In this course, you will learn more about sequence models, one of the most promising areas of deep learning. Models like recurrent neural networks (RNNs) have transformed speech recognition, natural language processing and other areas. There are some examples in the image below (the sequences can be found either in the inputs either in the outputs).

Speech recognition		→	
Music generation	∅	→	
Sentiment classification	"There is nothing to like in this movie."	→	★☆☆☆☆
DNA sequence analysis	AGCCCCTGTGAGGAAGTAG	→	AGCCCCTGTGAGGAAGTAG
Machine translation	Voulez-vous chanter avec moi?	→	Do you want to sing with me?
Video activity recognition		→	Running
Name entity recognition	Yesterday, Harry Potter met Hermione Granger.	→	Yesterday, Harry Potter met Hermione Granger .

Notation

Let's start defining a notation that we'll use to build up these sequence models. We'll start with a motivating example of a name entity recognition application:

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

$x^{<1>} \quad x^{<2>} \quad x^{<3>} \quad x^{<4>} \quad \dots \quad x^{<t>} \quad \dots \quad x^{<9>}$

$T_x = 9 \rightarrow$ length of the input sequence

y : $1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0$
 $y^{<1>} \quad y^{<2>} \quad y^{<3>} \quad y^{<4>} \quad \dots \quad y^{<t>} \quad \dots \quad y^{<9>}$

$T_y = 9 \rightarrow$ length of the output sequence

$X^{(i)<t>}$: the element # t in the input sequence of the training example # i

$T_x^{(i)}$: length of the input training example # i

$Y^{(i)<t>}$: the element # t in the output sequence of the training example # i

$T_y^{(i)}$: length of the output training example # i

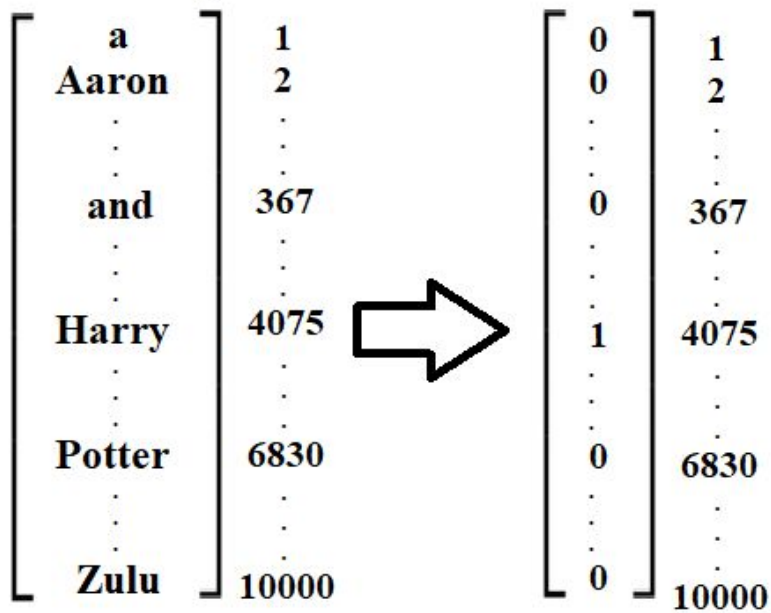
To represent a word in the sentence, the first thing you do is come up with a vocabulary (sometimes also called a dictionary), and that means making a list of the words that you will use in your representations. Let's see an example:

Vocabulary

a	1
Aaron	2
.	.
.	.
and	367
.	.
.	.
Harry	4075
.	.
.	.
Potter	6830
.	.
.	.
Zulu	10000

In this case, we've chosen a dictionary with 10k words (commercial applications usually use dictionaries with around 30-50k words). To represent a word we could use the one-hot representation. If we want to represent the word Harry, for example:

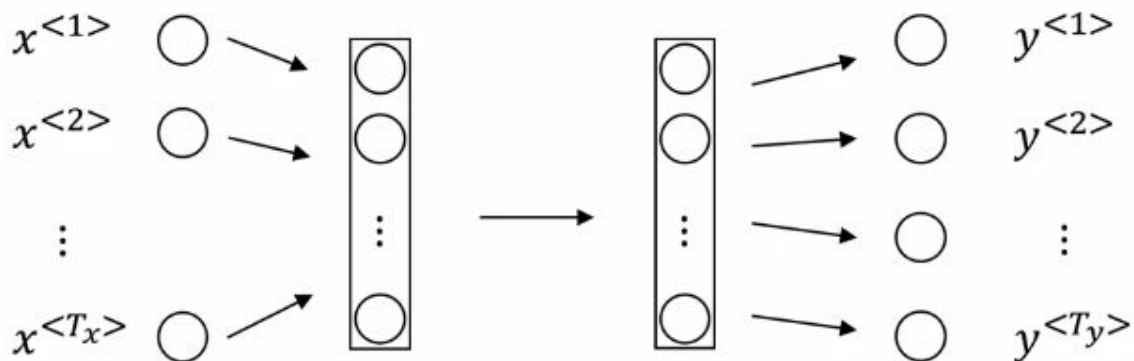
Vocabulary

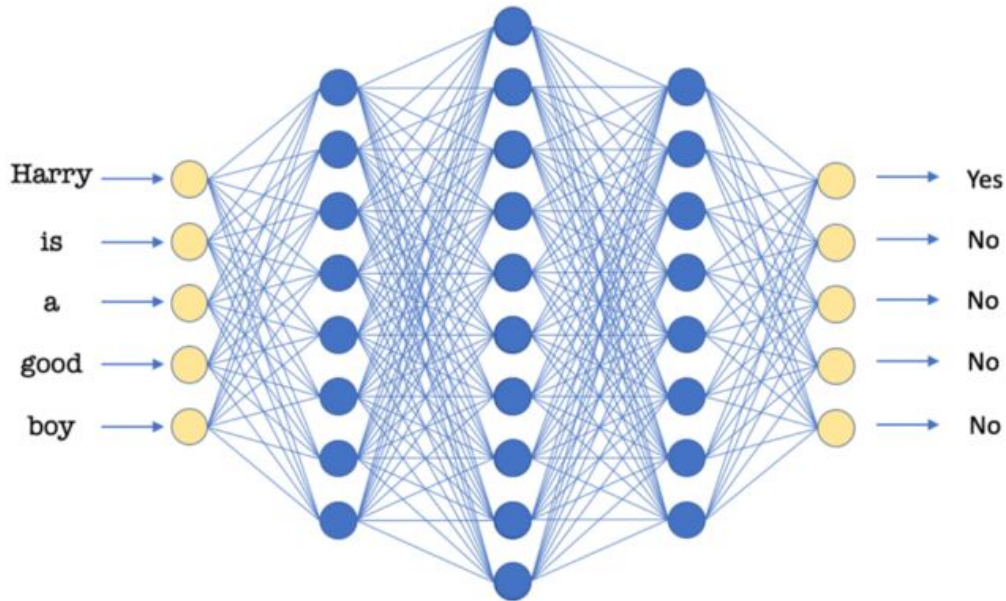


Harry is the element $x^{<1>}$ of the input sequence, so it is represented by a vector of 10000 elements, with only one element equals to 1 and the rest are 0's.

Recurrent Neural Network Model

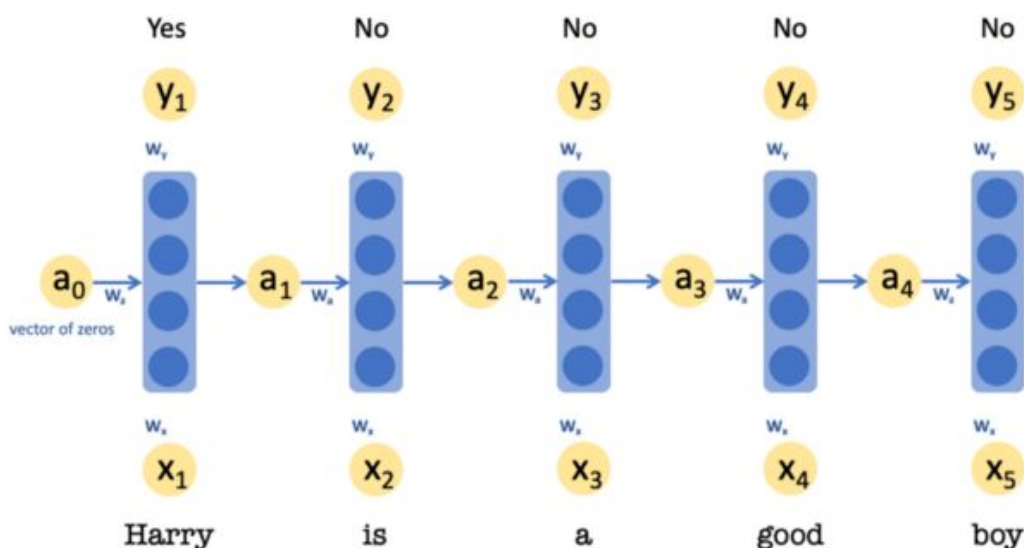
With the notation of sequence models learned, let's now build a neural network model to learn mapping from x to y . One question you could think is: why not to use a standard network like these ones below? (using again the name entity recognition example)





However, there are some problems with the standard models when using sequence models. One of the problems is that the inputs and outputs can be different lengths in different examples. The other problem is that it doesn't share features learned across different positions of text.

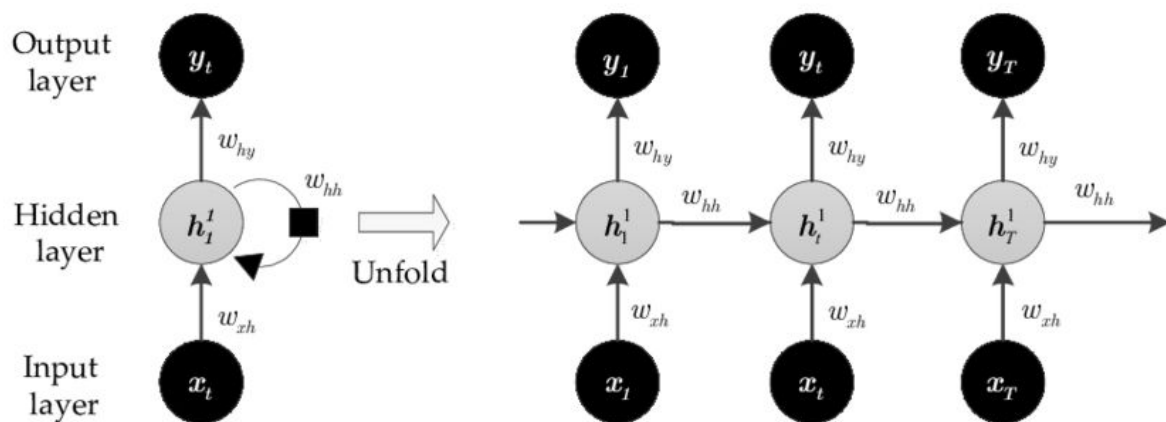
In Recurrent Neural Networks, each word is passed through a hidden layer. This hidden layer produces two vectors for each word - an output vector and an activation vector. The activation vector from the previous word is also used in conjunction with the current word. These two combined, produce an output vector for the current word and an activation vector which will be used along with the next word. This process is continued till the end of the sentence.



The Recurrent Neural Network scans through the data from the left to the right and the same set of parameters are used for each word of the sequence (the parameters are w_{aa} , w_{ya} and w_{ax}). So, the above RNN, while making the prediction for a

particular word, not only gets information from that word but also from the previous words which is passed on to it by the activation vector.

There are two different ways to represent a recurrent neural network: the folded version and the unfolded version. You can see both of them in the picture below, where the folded version is the left one, and the unfolded version is on the right:



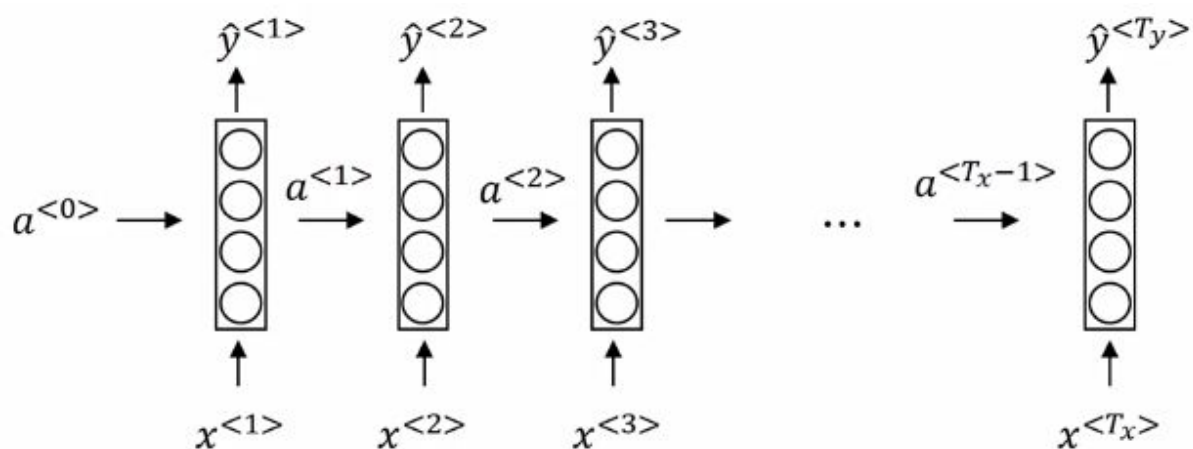
There is a slight problem with only using the previous hidden layers' information. The problem is exemplified in the following sentences:

"Teddy Roosevelt was a great president."

"Teddy bears are on sale!"

In this case, it's not possible to determine if the word "Teddy" means the name of a person or a toy. So it's necessary to have information about the next hidden layers as well, not only the previous ones. This problem is solved using bidirectional RNNs, that we'll talk about in the future sections.

Let's learn how to do forward propagation:



$$a^{<0>} = 0$$

$$a^{<1>} = g_1(w_{aa} a^{<0>} + w_{ax} x^{<1>} + b_a)$$

$$\hat{y}^{<1>} = g_2(w_{ya} a^{<1>} + b_y)$$

...

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

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

In order to help us develop more complex neural networks, let's simplify this notation a bit.

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

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

The matrix w_a is just the matrices w_{aa} and w_{ax} stacked horizontally. The notation $[a^{<t-1>}, x^{<t>}]$ denotes that we are going to take the two vectors and stack them together.

$$w_a = \begin{bmatrix} w_{aa} & w_{ax} \end{bmatrix} \quad [a^{<t-1>}, x^{<t>}] = \begin{bmatrix} a^{<t-1>} \\ x^{<t>} \end{bmatrix}$$

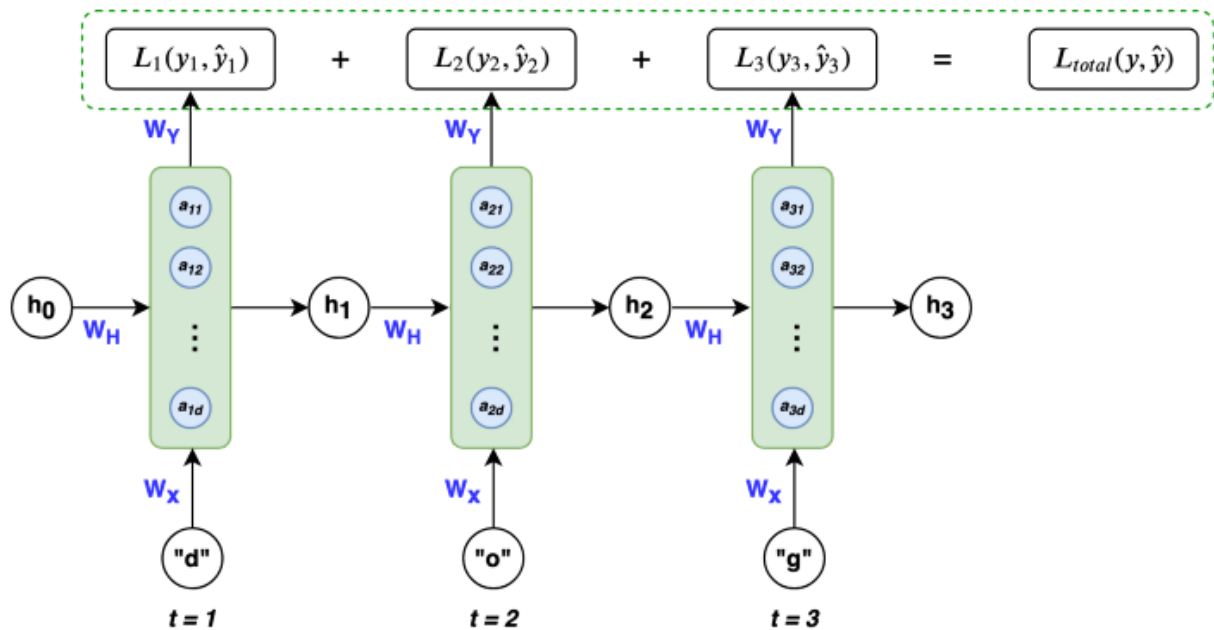
So $w_a [a^{<t-1>}, x^{<t>}] = w_{aa} a^{<t-1>} + w_{ax} x^{<t>}$.

Backpropagation through time

In this section, we'll learn how backpropagation works. The general workflow of a recurrent neural network is the following:

1. Initialize weight matrices W_x , W_y and W_h randomly
2. Forward propagation to compute predictions
3. Compute the loss
4. Backpropagation to compute gradients
5. Update weights based on gradients
6. Repeat steps 2-5

We'll use an example of recognizing the word "dog". The architecture is the following:



In this case, we'll use the cross-entropy loss (the same one used in logistic regression). For each timestep, we'll compute a loss and then sum all the losses to compute the total loss.

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

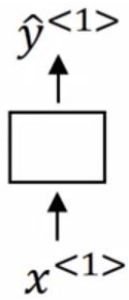
$$L(\hat{y}, y) = \sum_{t=1}^{T_y} L^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

Analysing the total loss allows us to know if our algorithm is performing well or not. We'll discuss later how to compute the gradients of the weight matrices to update the parameters.

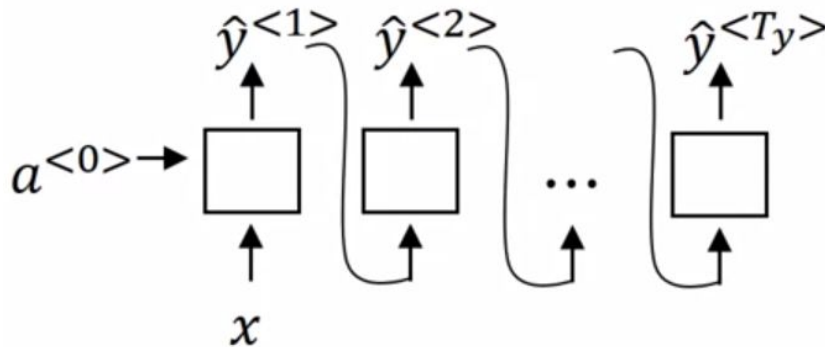
Different types of RNNs

So far, we've seen an RNN architecture where the number of inputs T_x is equal to the number of outputs T_y . However, for other applications, T_x and T_y may not always be the same. Let's look some examples:

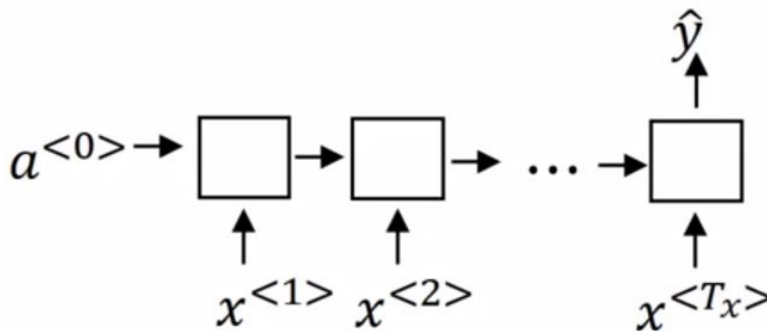
1 - One to one RNN: One to One RNN are the most basic and traditional type of recurrent neural network. Given a single input, there will be only one output as well ($T_x = T_y = 1$).



2 - One to Many: One to Many is a kind of RNN architecture that is applied in situations that one single input results in more than one output. One application of this architecture is in music generation ($T_x = 1, T_y > 1$).

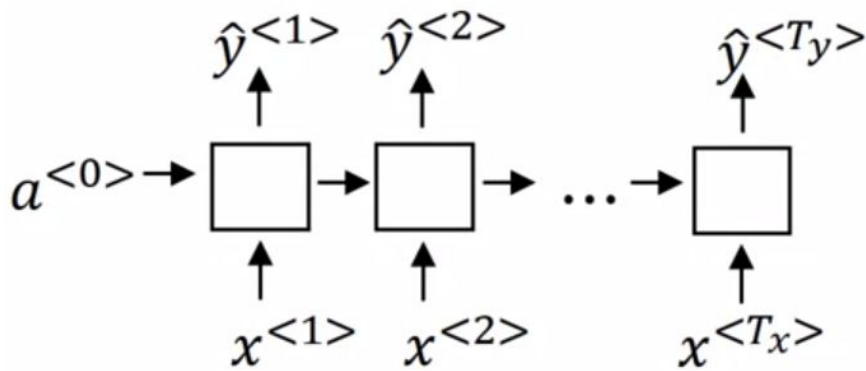


3 - Many to One: Many to One architecture is a kind of RNN architecture that is applied in situations where more than one input results in a single output. It is usually used in sentiment analysis and movie rating models that take review texts as input and provide a rating to a movie that may range from 1 to 5 stars ($T_x > 1, T_y = 1$).

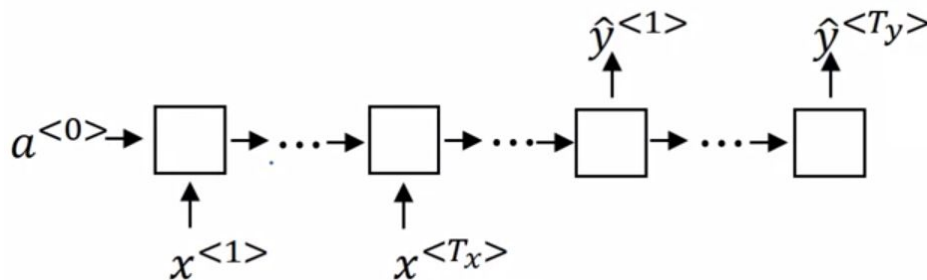


4 - Many to Many: Many to Many architecture takes multiple inputs and gives multiple outputs. Many to Many is divided into two types:

4.1 - $T_x = T_y$: For every input there is also an output. A common use for this type of RNN is in name-entity recognition.



4.2 - $T_x \neq T_y$: The inputs and outputs are different sizes. The most common application of this kind of RNN architecture is machine translation. For example, “the book is on the table” is an english sentence of 6 words, and is translated in portuguese “o livro está na mesa”, that is a sentence with 5 words.



In the machine translation application, the left blocks in this image are called “encoder” and the right blocks are called “decoder”.

Language model and sequence generation

Language modeling is one of the most basic and important tasks in natural language processing. Let’s say you are building a speech recognition system and you hear the sentence “The apple and pear salad was delicious”. Your system might not understand the correct sentence in the first try, because it can understand two different sentences:

“The apple and pair salad.”

“The apple and pear salad.”

So your job is to produce a good model that recognizes only the second sentence as the correct one, even if the two sentences are pronounced the same way. So what the model language does, is given any sentence, it computes the probability of different sentences being the correct one. Using the previous sentences as an demonstration:

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

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

In this case, the biggest probability represents the recognized sentence by the system. The basic job of a language model is to input a sentence, which is

represented by the sequence $y^{<1>}, y^{<2>}, \dots, y^{<T_x>}$ (for language models it will be useful to represent a sentence as outputs y rather than inputs x) and estimates the probability of that particular sequence of words. To build a language model with a RNN, you'll need:

Training set: large corpus of english text (corpus is a NLP terminology that means a large body/very large set)

For example:

“Cats average 15 hours of sleep a day.”

We need first to tokenize the sentence. For language models it's very useful to append an <EOS> (end of sentence) token in the sentence, so the model knows where one sentence ends and where another sentence begins.

“Cats average 15 hours of sleep a day <EOS>”

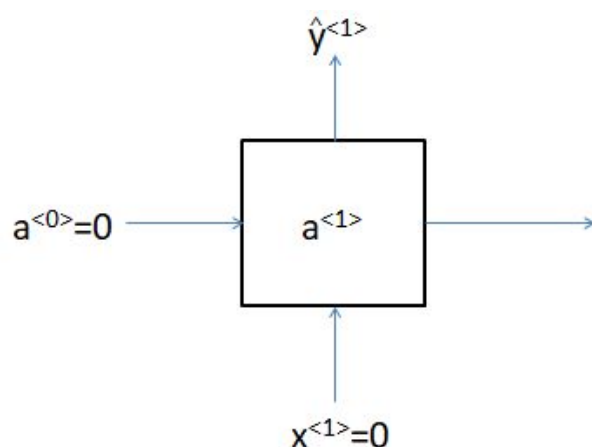
$y^{<1>} \quad y^{<2>} \quad y^{<3>} \quad y^{<4>} \quad y^{<5>} \quad y^{<6>} \quad y^{<7>} \quad y^{<8>} \quad y^{<9>}$

Another example:

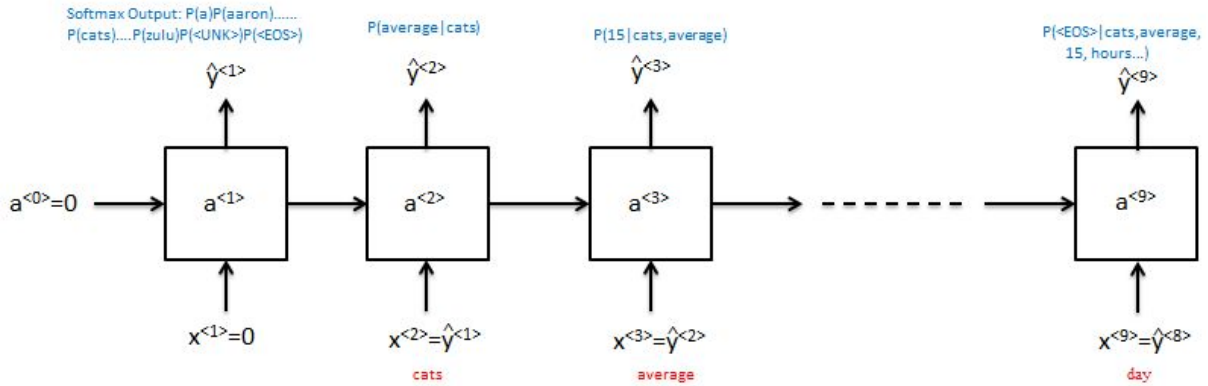
“The Egyptian Mau is a breed of cat <EOS>”

In this example, the word Mau is a very uncommon word, so if your vocabulary uses around 10k words (the 10k most common english words), then the word Mau might not be in the vocabulary. In this case, you could take the word Mau and replace it with a unique token called UNK (unknown word). The model would have to compute the chances of this unknown word instead of the specific word “now”.

Let's build the model using the “Cats average 15 hours of sleep a day <EOS>” sentence. The inputs to the first RNN unit $x^{<0>}$ and $a^{<0>}$ are a vector of zeros. $\hat{y}^{<1>}$ is the softmax probabilities of each word in the dictionary to be in that position of the sentence.



The output of the first unit is passed on as an input to the second unit -> $x^{<2>} = \hat{y}^{<1>}$. Similarly, $x^{<3>} = \hat{y}^{<2>}$ and so on.



$$P(\text{sentence}) = P(\hat{y}^{<1>}, \hat{y}^{<2>}, \hat{y}^{<3>}, \dots) = P(\hat{y}^{<1>}) P(\hat{y}^{<2>} | \hat{y}^{<1>}) P(\hat{y}^{<3>} | \hat{y}^{<1>}, \hat{y}^{<2>}) \dots$$

The $P(\text{sentence})$ is the probability of the sentence to have the words “Cats average 15...” and is defined by the product of each individual probability. The cost function used in this case is the softmax cost function:

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

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

Our goal is to reduce the loss.

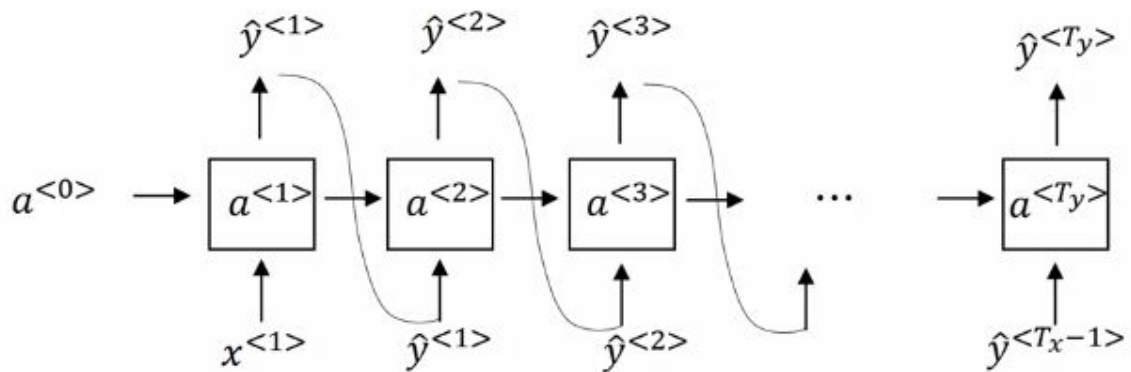
Sampling novel sequences

After training a sequence model, one of the ways you can informally get a sense of what is learned is to have sample novel sequences. First of all, you are going to sample what is the first word you want your model to generate. As usual, the input x_1 and the activation vector a_0 are equal to 0. The first time stamp will be randomly sampled according to the softmax distribution. For example:

Let's say you have a vocabulary with 10k English words. Each one of these words will have a probability of being the first word of the sentence and to choose one of these words you need to use `np.random.choice` to randomly select one word in that distribution.

The sampled word in the first unit is then passed as an input to the second unit. The process of sampling is repeated until a complete sentence is formed.

Vocabulary = [a, aaron, ..., zulu, <UNK>]



You can also use some character level model and instead of using a vocabulary of words, you can use a vocabulary of single characters, especially if you have a lot of computational power.

Vocabulary = [a, b, c,..., z, , . , , ; , : , 0, ..., 0, A, ... , Z]

Examples of sequence generation:

News

President enrique peña nieto, announced
sench's sulk former coming football langston
paring.

"I was not at all surprised," said hich langston.

"Concussion epidemic", to be examined.

The gray football the told some and this has on
the uefa icon, should money as.

Shakespeare

The mortal moon hath her eclipse in love.

And subject of this thou art another this fold.

When besser be my love to me see sabl's.

For whose are ruse of mine eyes heaves.

Vanishing gradients with RNNs

As you probably remember from previous courses, the gradient descent algorithm finds the global minimum of the cost function that is going to be an optimal setup for the network. In traditional networks, information travels through the neural network from input neurons to the output neurons, while the error is calculated and propagated back through the network to update the weights. It works quite similarly for RNNs, but here we've got a little bit more going on.

-Firstly, information travels through time in RNNs, which means that information from previous time points is used as input for the next time points.

-Secondly, you can calculate the cost function, or your error, at each time point.

Essentially, every single neuron that participated in the calculation of the output, associated with the cost function, should have its weight updated in order to

minimize that error. The problem with RNNs is that it's not just the neurons directly below this output that contributed but all of the neurons far back in time. So you have to propagate all the way back through time to these neurons.

Weights are assigned at the start of the neural network with random values, which are close to zero, and from there the network trains them up. But, when you start with the weights close to zero and multiply the inputs by this value, your gradient becomes less and less with each multiplication and the lower the gradient is, the harder it is for the network to update the weights and longer it takes to get to the final result.

There is also the exploding gradient problem that happens when the gradient gets a very large number.

Solutions to the vanishing gradient problem:

- Use GRU: gated recurrent unit, which is a very effective solution for the vanishing gradient problem;
- Initialize weights so that the potential for vanishing gradient is minimized;
- Have Echo State Networks that are designed to solve the gradient problem;
- Have Long Short-Term Memory Networks (LSTMs).

Solutions to the exploding gradient problem:

- Gradient Clipping: Put a maximum limit on a gradient (some threshold value) and if the gradient is bigger than the threshold, re-scale some of your gradient vector so that it is not too big;
- Stop backpropagating after a certain point, which is usually not optimal because not all of the weights get updated;
- Penalize or artificially reduce gradient.

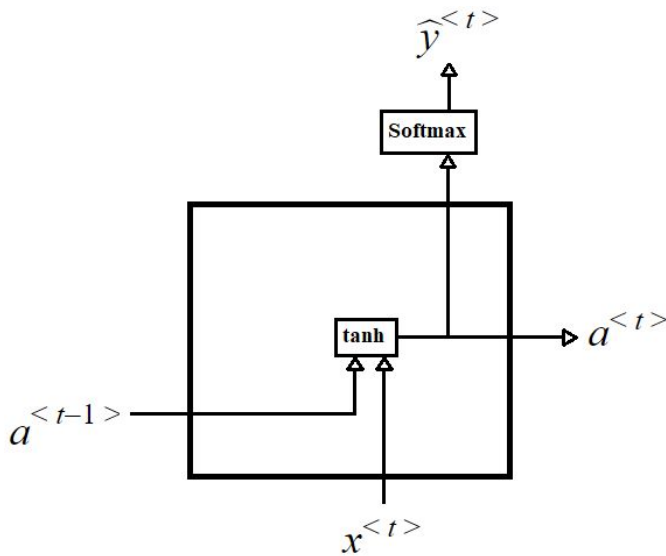
Gated Recurrent Unit (GRU)

The gated recurrent unit 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.

First, let's remember the traditional model for a RNN unit:

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

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



Now let's show a sentence example:

"The cat, which already ate, ..., was full. "

In order to keep the verbal agreement between the subject "cat" and the verb "was", we'll add a memory cell variable "c" to the model, so we can keep the subject saved no matter what is the size of the sentence. At some time t the memory cell c will have some value $c^{<t>}$. The GRU unit will actually output an activation value $a^{<t>}$, that is going to be the value of $c^{<t>}$ ($a^{<t>} = c^{<t>}$). Obs: for the LSTM models, $a^{<t>} \neq c^{<t>}$, but for now they are the same value. The candidate value equation for $\tilde{c}^{<t>}$ is:

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

With this candidate value we'll define the actual value for the cell variable. Let's also define the gate variable Γ_u . This gate value is usually always zero or one, because it's defined as a sigmoid function.

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

The subscript "u" stands for "update".

Imagine we set the variable $c^{<t>}$ to be equals 0 when the word is in the plural form and 1 when it is in the singular form. The sentence is going to be read word by word and when it is needed to check the verbal agreement of some verb, the model will use the memory cell and check if the previous subject word was a singular or plural word (using "was" if singular or "were" if plural).

Then, the actual value of $c^{<t>}$ is:

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

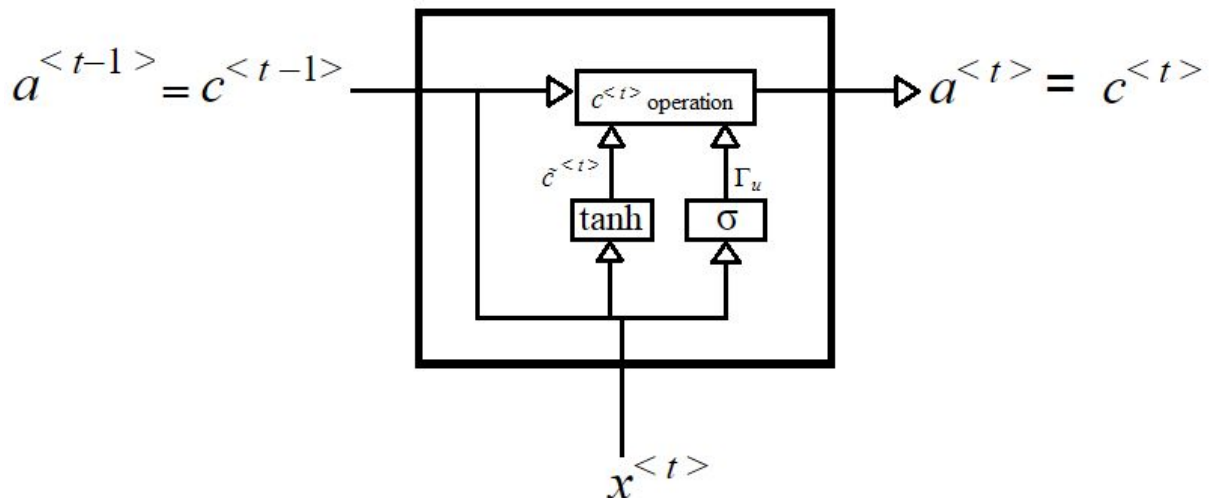
If $\Gamma_u = 0$, the value of $c^{<t>}$ is going to be equals the value of the previous candidate $\tilde{c}^{<t-1>}$. If $\Gamma_u = 1$, the value of $c^{<t>}$ will change to the actual value of $\tilde{c}^{<t>}$. If you use the sentence "The cat, which already ate, ..., was full. " for example, the parameter Γ_u will be equals to one when it is in the word "cat", 0 for all

the other words and 1 again when it is in the word “was”. In this case you carry the singular word “cat” throughout the whole sentence until you find the verb related to this word.

“The cat, which already ate, ..., was full. “

$$\Gamma_u = 1 \quad \Gamma_u = 0 \quad \Gamma_u = 0 \quad \Gamma_u = 0, \dots, \Gamma_u = 1$$

The new block diagram is going to be like this:



$c^{<t>}$ can be a vector with a large number of elements. If $c^{<t>}$ is a vector, $\dim c^{<t>} = \dim \tilde{c}^{<t>} = \dim \Gamma_u$. What we described is actually a simple GR unit, let's describe now the full GRU, that adds a new parameter Γ_r .

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

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

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

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

The subscript “r” stands for “relevance”. The gate Γ_r tells you how relevant $c^{<t-1>}$ is to compute the next candidate for $\tilde{c}^{<t>}$. A different version of these equations form the LSTM networks. Obs: if you look for academic papers, the variables can have different names -> $\tilde{c} = \tilde{h}$, $\Gamma_u = u$, $\Gamma_r = r$, $c = h$.

Long Short Term Memory (LSTM)

The LSTM is an even more powerful tool that allows you to learn very long range connections in a sequence. Let's compare the GRU and the LSTM models. Below you can see the GRU equations:

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

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

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

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

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

And the LSTM equations:

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

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

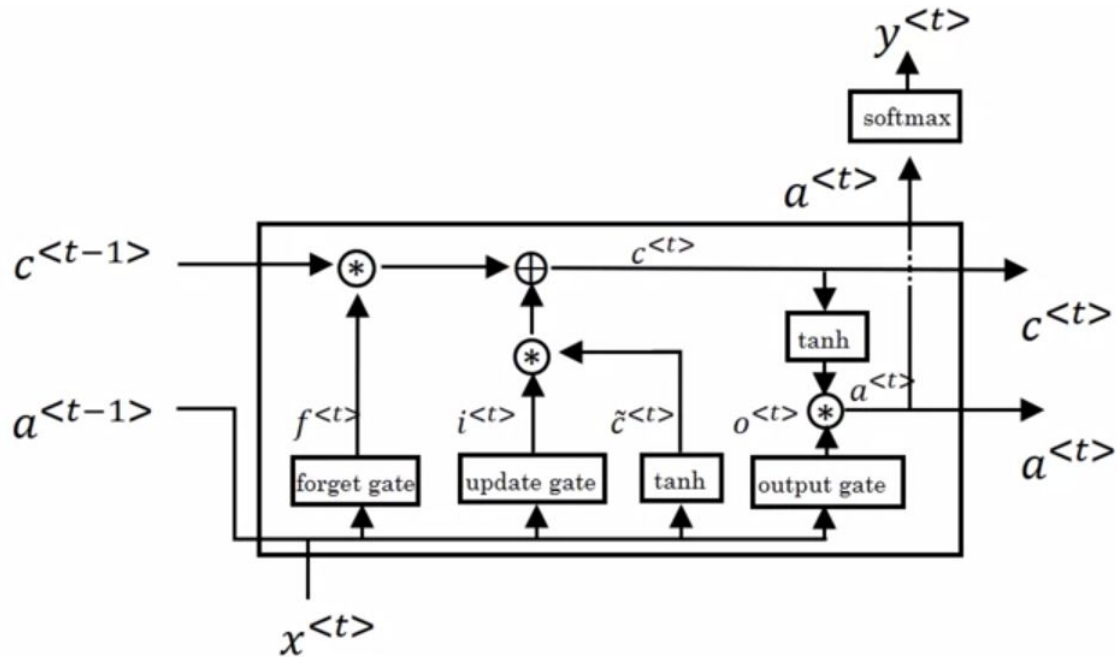
$$\Gamma_f = \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f)$$

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

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

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

“u” stands for “update”, “f” for “forget” and “o” for “output”. The block diagram can be seen in the picture below:

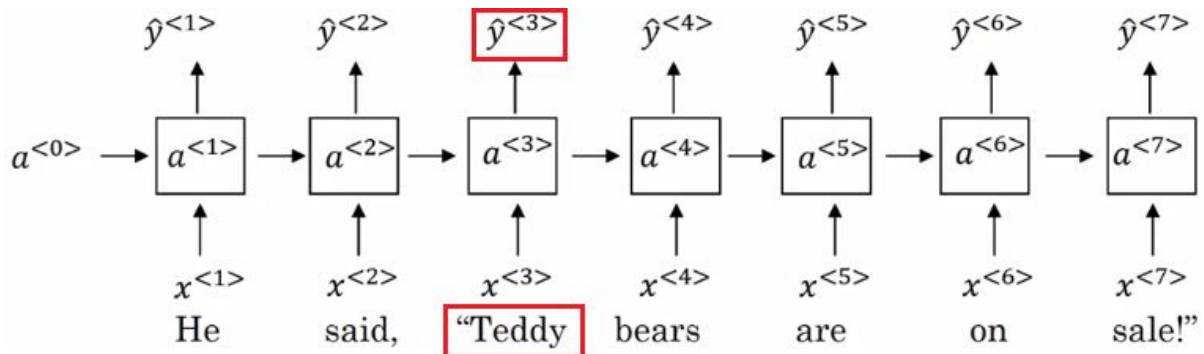


There is not a consensus on when to use the GRU or when to use the LSTM models. The advantage of the GRU is that it's a simpler model and is actually easier to build a much bigger network, and with only two gates it runs a bit faster. However,

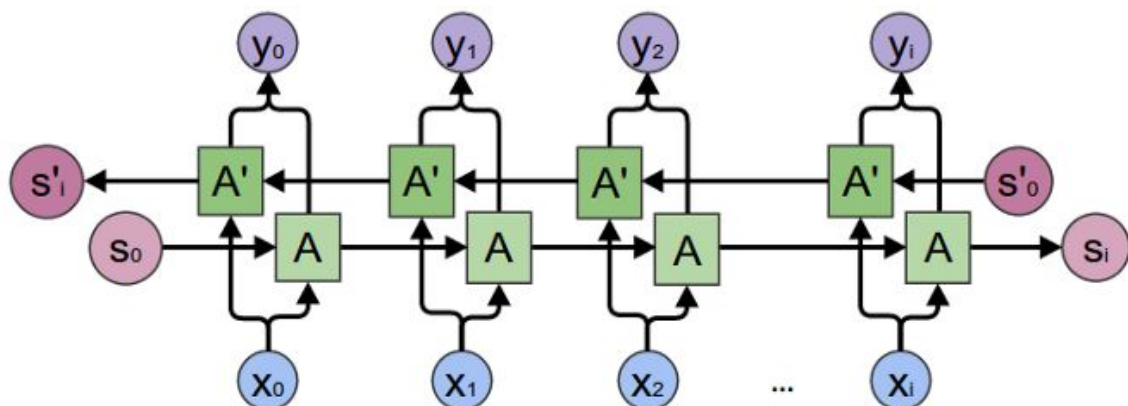
the LSTM is more powerful and more effective since it has three gates instead of two. If you have to choose, you might try the LSTM model first, but both of them should work just fine.

Bidirectional RNN

Bidirectional RNNs let you at a point in time to take information from both earlier and later in the sequence. Let's study an example:



In this case, using only a directional RNN it's not possible to know if Teddy is the name of a person or a toy. To tell if $\hat{y}^{<3>}$ should be 0 or 1, you need more information than just the three words of the sentence (no matter if you are using standard RNN units, GRU or LSTM). The bidirectional RNN (BRNN) solves this problem, let's see its representation:



The bidirectional RNN defines an acyclic graph. The forward propagation is composed of two parts, one goes from the left to the right and the other one goes from the right to the left.

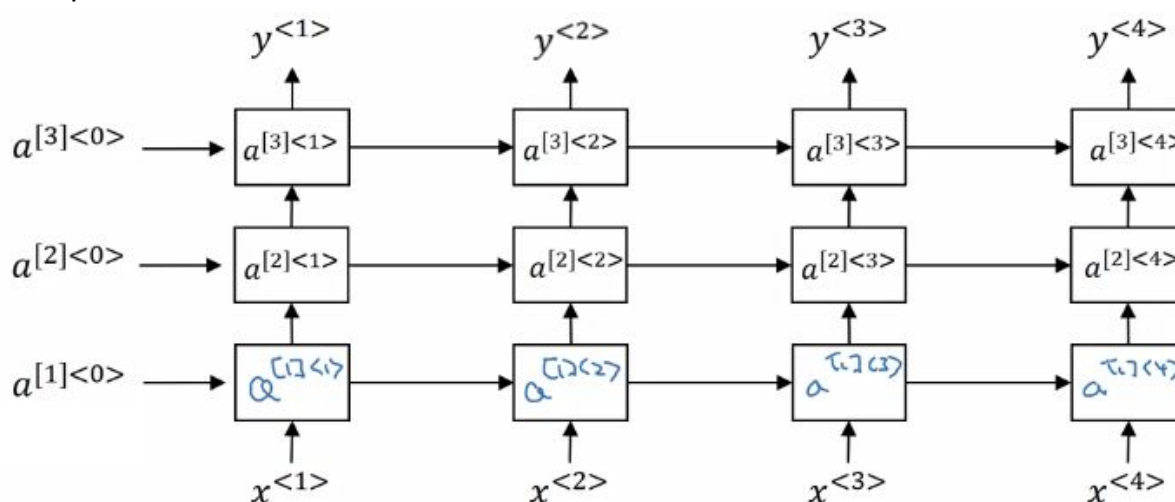
$$\hat{y}^{<t>} = g(w_y [A, A'] + b_y)$$

If you look at the prediction at time step three, information from the two first units and information from the fourth unit is passed to the third unit, so it allows the unit to know that Teddy stands for a toy, not a part of a persons' name. The advantage of BRNNs is that it's able to make predictions anywhere even in the middle of a sequence by taking into account information potentially from the entire sequence.

The disadvantage is that you do need the entire sequence of data before you can make predictions and if you are building a speech recognition system for example, you'll have to wait for the person to stop talking before you can actually process it and make a speech recognition prediction.

Deep RNNs

For learning very complex functions sometimes it is useful to stack multiple layers of RNNs together to build even deeper versions of these models. You can see an example below:



As you noticed, we added a new superscript to the activations, to represent the corresponding layer. Let's make an analysis of the $a^{[2]<3>}$ activation function:

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

As you can see, the values of the third unit from the first layer and the second unit from the second layer are used to compute the activation value of $a^{[2]<3>}$.

We've seen neural networks with more than 100 layers, but for RNNs having three layers is already quite a lot because of the temporal dimension these networks can get very big. You can see some deep networks connected to the outputs of each unit, but they will not be connected horizontally. In bidirectional RNNs, each unit can use standard RNN blocks, GRU blocks or even LSTM blocks.

NLP & Word Embeddings - Introduction to Word Embeddings

Word Representation

Let's assume we are using a vocabulary of 10000 words. To represent each one of these words, we use the 1-hot representation, it means that there will be a vector of size 10000 for each word, with 9999 elements equal to zero and one equals 1. We'll also add a notation to represent the words, for example imagine that the word "Man"

is the word number 5791 in the vector, we are going to represent this word as O_{5791} .
For example:

Man (5391)	Woman (9853)	King (4914)	Queen (7157)	Apple (456)	Orange (6257)
$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}$	$\begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}$
O_{5391}	O_{9853}	O_{4914}	O_{7157}	O_{456}	O_{6257}

However, the 1-hot representation is not very useful in some applications. For example:

“ I want a glass of orange _____ . “

“ I want a glass of apple _____ . “

The word “juice” correctly fills both of these examples, however if the learning algorithm has learned that “I want a glass of orange juice” is a likely sentence, if it sees “I want a glass of apple _____ . “ there is no defined relationship between apple and orange, so the algorithm will not know what word to use to fill the blank (the probability of using the words “juice” or “king” for example is the same). To solve this problem, instead of using an 1-hot representation, we’ll use a 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

In this case we used four different features, but we could use as many features as we need for our application to work well, such as size, cost, if it is a noun or a verb, etc.

The feature vector is associated with a variable called e_n , where n is the number of the word in the vocabulary. For example:

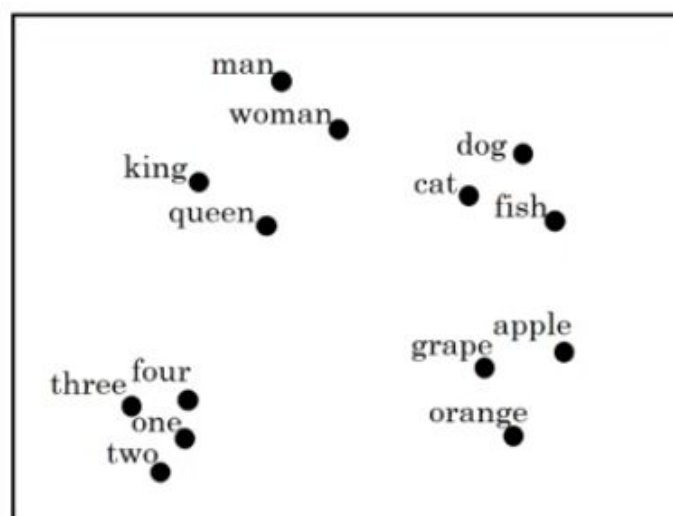
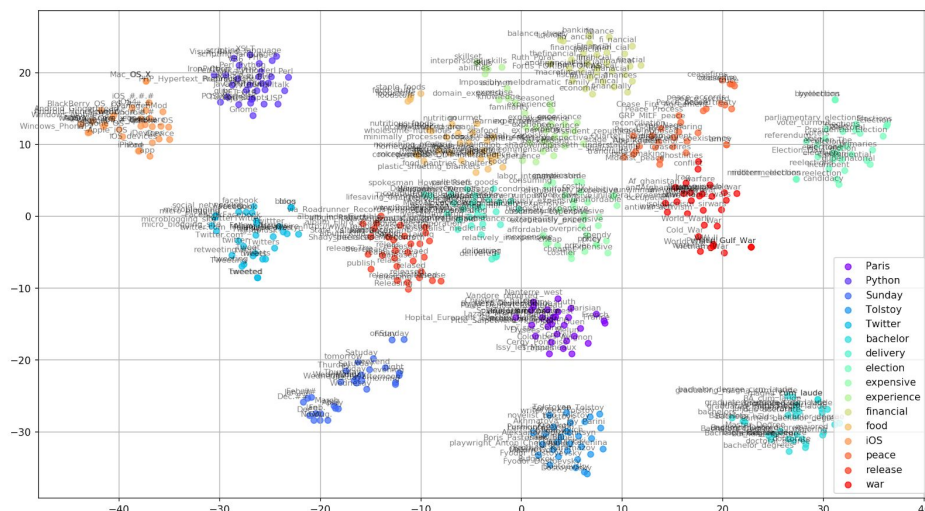
$e_{5391} = \text{feature vector containing } [-1, 0.01, 0.03, 0.04]$

Using the same sentences used in the 1-hot representation:

“ I want a glass of orange juice. “

“ I want a glass of apple _____ . “

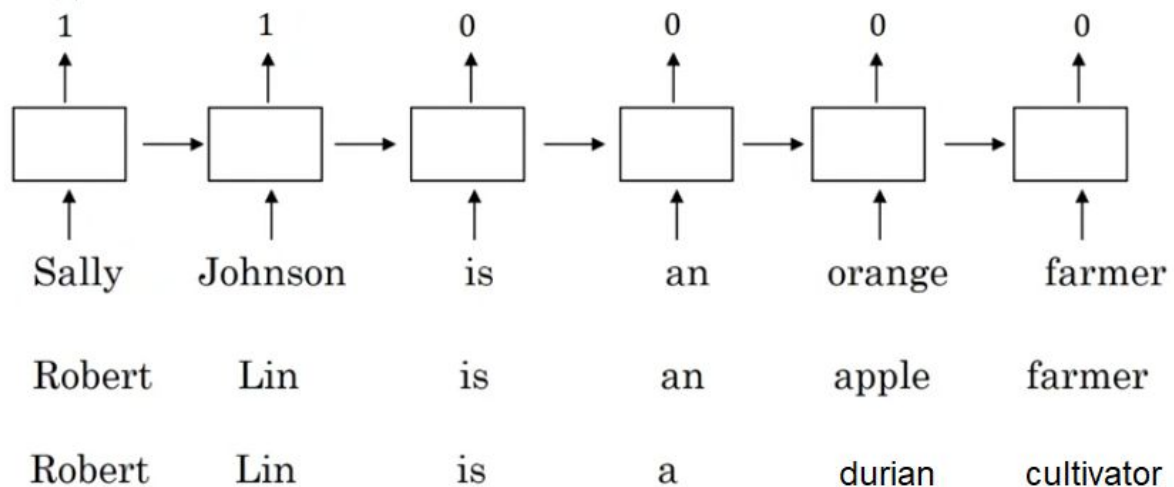
Notice that the representations of the words “orange” and “apple” are now quite similar. This form of representation increases the odds of the learning algorithm to figure out that if the word “juice” is a thing for “orange”, it might be as well for “apple”. If you are able to learn a 300 dimensional feature vector, one of the popular things to do is also to take this vector and embed it in a two dimensional space so you can visualize them. For example:



One popular algorithm of word embedding is the t-SNE algorithm (t-Distributed Stochastic Neighbor Embedding) developed by Laurens van der Maaten and Geoffrey Hinton.

Using Word Embeddings

In this section, we'll learn how to use the representations and plug them into NLP applications. Let's start with an example of named entity recognition.



If you train a model to recognize Sally Johnson as a person's name because of the "orange farmer" words, using the feature vector will also allow your model to recognize Robert Lin as a person's name, because "apple" has almost the same attributes as "orange".

However, if you try a rare fruit name, such as "durian" and a different profession like "cultivator", you need to make sure that these words are in your vocabulary and were used in the learning of the word embedding. Then, your model might be able to generalize from having seen an "orange farmer" in the training set to knowing that a "durian cultivator" is also a person.

One of the reasons that word embeddings are able to do complex associations is that the learning process of word embeddings can use and examine very large text corpuses, with billions of words. By examining tons of unlabeled text, which you can download for free on the internet, you can figure out that orange and durian are similar, and farmer and cultivator are also similar. Then, you can take this word embedding and apply it to your name recognition task (use transfer learning), for which you might have a much smaller training set, maybe with just 100k words (or even smaller).

Let's do an algorithm to apply transfer learning:

1. Learn word embeddings from large text corpus. (1-100B words) (Or download pre-trained embedding online.)
2. Transfer embedding to a new task with smaller training set.
3. Optional: Continue to finetune the word embeddings with new data.

Properties of word embeddings

One of the most fascinating properties of word embeddings is that they can also help with analogy reasoning. Let's see an 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

With this table feature representation, we want to know the following dilemma:

Man is to woman as king is to ????

You probably straight answered "queen", but is it possible to have an algorithm figure this out automatically?

For learning purposes, let's substitute the notation e_{number} with e_{name} . For example, instead of using e_{5391} , we'll use e_{man} . If you take the vector e_{man} and subtract it with the vector e_{woman} , you'll have the following results:

$$[-1, 0.01, 0.03, 0.09] - [1, 0.02, 0.02, 0.01] \approx [-2, 0, 0, 0]$$

Similarly, if you take $e_{king} - e_{queen}$, you'll have:

$$[-0.95, 0.93, 0.70, 0.02] - [0.97, 0.95, 0.69, 0.01] \approx [-2, 0, 0, 0]$$

What these operations are capturing is that the main difference between man and woman is the gender, and the main difference between king and queen is also the gender. So to answer the question, "man is to woman as king is to ???", what you can do is to compute $e_{man} - e_{woman}$ and try to find a word so that $e_{king} - e_{word}$ is close to $e_{man} - e_{woman}$. It turns out that when queen is the word plugged, then the difference vectors practically match with each other.

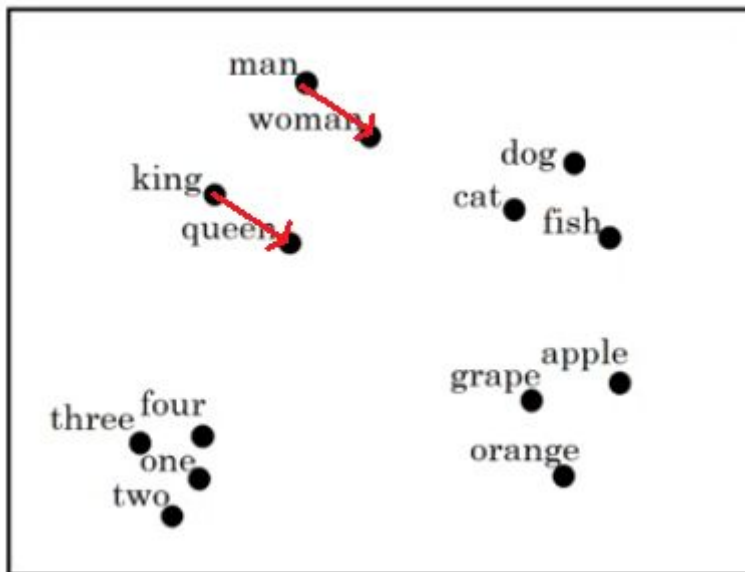
These ideas were presented by Mikolov in 2013, in the paper "Linguistic regularities in continuous space word representations".

If we have 300 features, the vector will be 300 dimensional. In this case, we want to find a word w that respects the following condition:

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

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

The objective is to find a word that maximizes the similarities between the word and the rest of the expression ($e_w \approx e_{king} - e_{man} + e_{woman}$). In this case, the word that maximizes the similarities is the word “queen”.



Given two vectors, A and B, the similarity between them can be computed with the following formula:

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}$$

Where θ is the angle between the vectors. You can see some examples below of what word embedding learning can associate:

Man:Woman as Boy:Girl (gender)

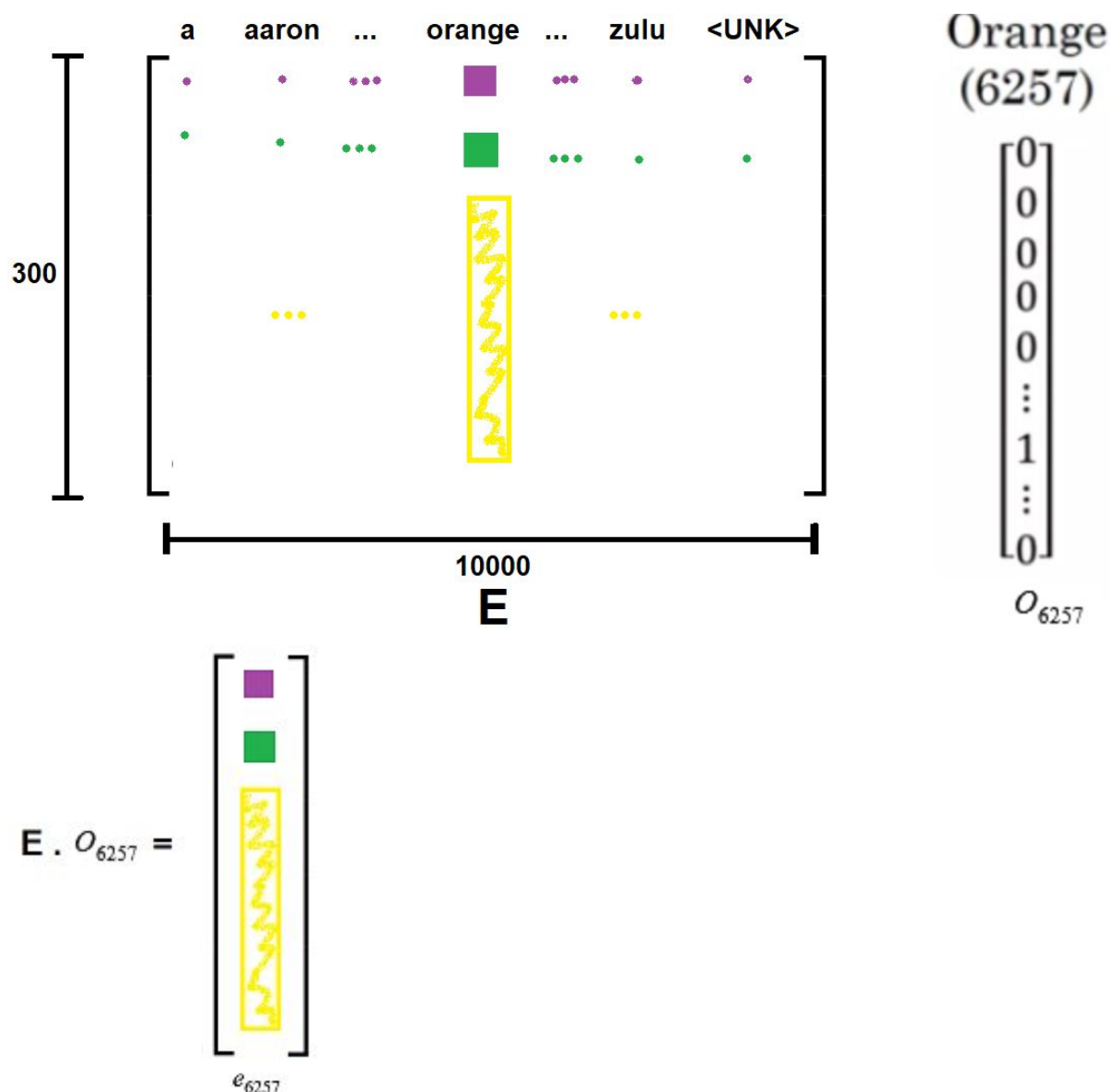
Ottawa:Canada as Nairobi:Kenya (capital and country)

Big:Bigger as Tall:Taller (comparative adjectives)

Yen:Japan as Ruble:Russia (currency and country)

Embedding matrix

When you implement an algorithm to learn a word embedding, what you end up learning is an embedding matrix. Let's use an example with 10000 word vocabulary 300 different features. If we multiply the learning embedding matrix E with the one-hot vector of a word (let's use the word orange), we will have the following:



In this case, only the column of the word “orange” is multiplied by 1 and all the other columns are multiplied by 0. This means that only the features of the desired word are actually preserved. More generally:

$$E \cdot O_j = e_j = \text{embedding for word } j$$

In practice the traditional matrix multiplication with the one-hot vector is not very effective because it wastes a lot of computational power multiplying huge amounts of numbers by zero. What is actually done is to use a specialized function to just look up a column of the matrix E rather than the whole matrix. Using keras for example there is the embedding layer (from the embedding class), that turns this task much easier.

Our goal is to learn an embedding matrix E . We’ll see more details about this matrix in the next sections.

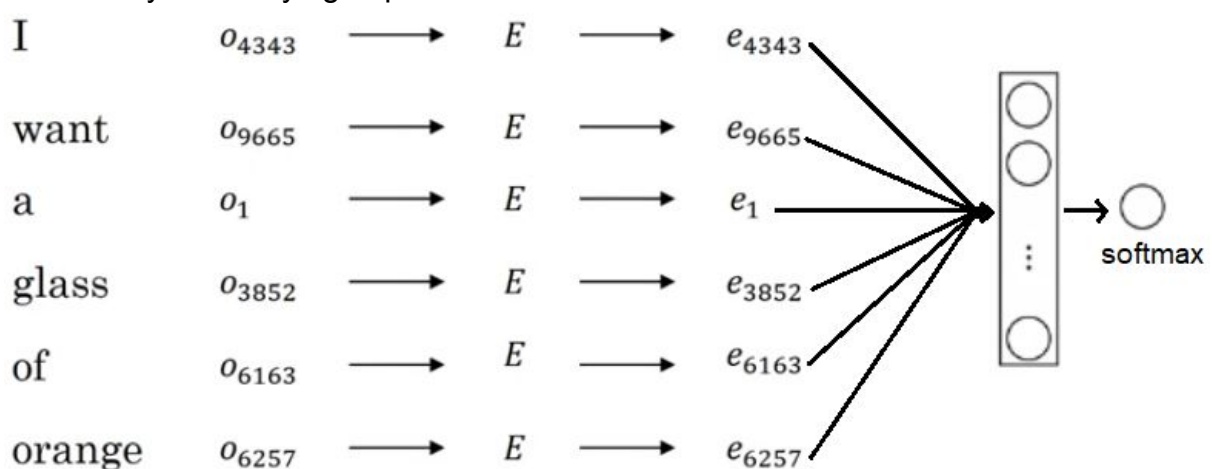
NLP & Word Embeddings - Learning Word Embeddings: Word2vec & GloVe

Learning word embeddings

In the history of deep learning, people actually started using relatively complex algorithms and then over time researchers discovered they can use simpler and simpler algorithms and still get very good results specially for large datasets. Let's learn more about these algorithms, using an example of word prediction (again with a 10000 word vocabulary and 300 features).

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

Now we'll build the neural language model for this problem. First of all, we are going to take all the one-hot vectors of the words in the sentence and multiply them by an embedding matrix. It will result in different embedding vectors. Then we fill all these embedding vectors in a neural network and the neural network feeds a softmax classifier, that classifies among the 10000 possible outputs in the vocabulary for the final word you are trying to predict.



The neural network and the softmax layers are going to have a set of parameters w and b , so the parameters are the matrix E and the weights and bias of the layers of the neural network. In this case, the input is going to be a $6 \times 300 = 1800$ vector (6 words with 300 features each). What is actually more used is to have a fixed historical window, so you might decide that you always want to predict the next word given the previous four words, for example (in this case, 4 is a hyperparameter of the algorithm).

Other context/target pairs can also be used to predict words. For example:

I want a glass of orange juice to go along with my cereal.

Imagine that you want to predict the word juice. You can use different contexts:

- Last 4 words (a glass of orange _____);
- 4 words on left & right (a glass of orange _____ to go along with);
- Last 1 word (orange _____);
- Nearby 1 word: for example, the word glass is near the word you are trying to predict, so you use it in your training (it is called skip-gram model, you'll learn more about it in the next section).

Word2Vec

In this section, you'll learn about the Word2Vec algorithm which is a simple and comfortably more efficient way to learn word embeddings. You can see more details about this algorithm in the paper "Mikolov et.al., 2013. Efficient estimation of word representations in vector space". Let's say you are given the following word in your training set:

"I want a glass of orange juice to go along with my cereal."

In the skip-gram model, that we are going to learn more about in this section, we are going to use a random word to be the context word (instead of using the last four words, or other contexts). What we are going to do is randomly pick a word to be the context word. Let's say we chose the word "orange".

Then we'll randomly pick another word within some window of the context word, and choose that word to be the target word. So maybe you pick the word "juice", the word "glass" or the word "my" to be the target word. This is not an easy learning problem, because if you want to discover 10 words given the word "orange" it could be a lot of different words.

The job of the skip-gram model is to use a content word c and predict a target word t .

x -----> y
 Content c ("orange" for example) -----> Target t ("juice")

Given:

O_c (One – hot vector of the content word)

E (Embedding matrix)

e_c (Embedding vector)

The overall model is going to be:

O_c -----> E -----> e_c -----> O -----> \hat{y}
softmax unit

$$e_c = E.O_c$$

The softmax model is going to be:

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

θ_t = parameter associated with output t

And the loss function of the softmax will be:

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

There are a couple of problems with using this model. The primary problem is computational speed, in particular for the softmax model, every time you want to evaluate the probability, you need to carry out a sum over all 10000 words in your vocabulary. To partially solve this problem, you can use a hierarchical softmax, that uses a tree to represent the words (you can read more about this technique in the paper cited in the beginning of the section).

One task we also need to do, is to sample the context c . One thing we could do is just sample uniformly at random from your training corpus. When we do that, we find that there are some words like “the”, “of”, “a”, “and”, “to”... that appear extremely frequently. However, in practice, the distribution of words isn’t taken just entirely at random from the training set, because the less common words (like “durian”) would rarely appear in the samples, so we would use different heuristics to balance out this problem.

Negative Sampling

In this section you’ll see a modified learning model called negative sampling, that allows you to do something similar to the skip-gram model but with a much more efficient learning algorithm. Most of the ideas presented in this section are due to the paper “Mikolov et.al., 2013. Distributed representation of words and phrases and their compositionality”. What we are going to do is, given a context word (like “orange” for example), we’ll randomly choose another word and check if it is a target word or not. For example:

<u>Context</u>	<u>word</u>	<u>target?</u>
orange	juice	1
orange	king	0
orange	book	0
orange	the	0
orange	of	0

To summarize, the way we generate this dataset is, we’ll pick a context word and then pick a target word and this will be the first row of the table. This process gives us a positive example. Then for some number of times (like k times), we’re going to

take the same context word and pick random words from the dictionary and label all those 0, and those will be our negative examples. The number of words randomly chosen is $k \approx 5-20$ in smaller datasets, and $k \approx 2-5$ in larger datasets.

We are going to create a supervised learning problem, where the learning algorithm inputs x (the context word and the positive/negative examples), and it has to predict the target label to predict the output y .

Then, we are going to define a logistic regression model:

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

It will work as a binary logistic regression classifier and in this example instead of training the whole vocabulary each iteration, we'll train only the target word and 4 more negative examples.

To summarize, instead of computing a softmax of 10000 words, we'll turn them into 10000 binary classification problems, that is quite cheap to compute, and on each iteration we're only going to train five of them (or generally k negative examples and one positive example).

Let's learn now how to randomly choose the negative examples. The probability of a word to be randomly chosen is defined by the following heuristic:

$$P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=1}^{10000} f(w_j)^{3/4}}$$

where $f(w_i)$ is the observed frequency of a particular word in the english language or in your training set corpus.

GloVe word vectors

The algorithm GloVe is not used as much as the Word2vec or the skip-gram models, but it has some enthusiasts in the NLP community, let's take a look at how it works. You can see more details about this algorithm in the paper "Pennington et. al., 2014. GloVe: Global vectors for word representation".

Previously, we were sampling pairs of words - context and target words - by picking two words that appear in close proximity to each other in our text corpus. What the GloVe algorithm does is, it starts off just by making this process explicit. Let's define a variable:

$X_{ij} = \#$ times that a word j appears in the context of i

In this case, i and j represent c and t respectively. So what the GloVe model does is optimize the following:

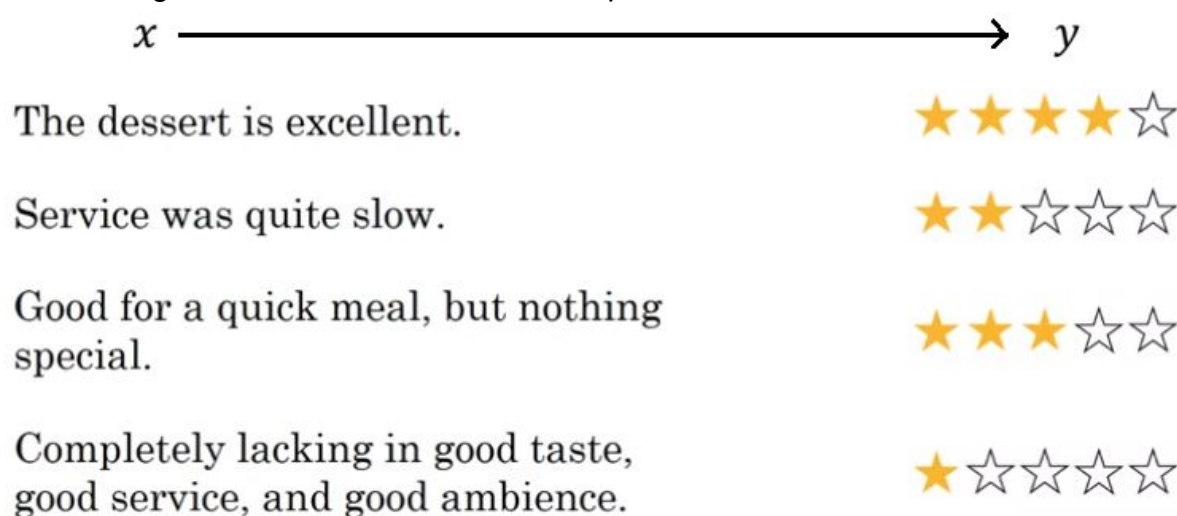
$$\text{minimize } \sum_{i=1}^{10,000} \sum_{j=1}^{10,000} f(X_{ij})(\theta_i^T e_j + b_i + b'_j - \log X_{ij})^2$$

In this function, $f(X_{ij})$ is a weighting term (if $X_{ij} = 0$, $f(X_{ij}) = 0$). This term allows us to not worry about the log term that would be a problem, because if $X_{ij} = 0$,

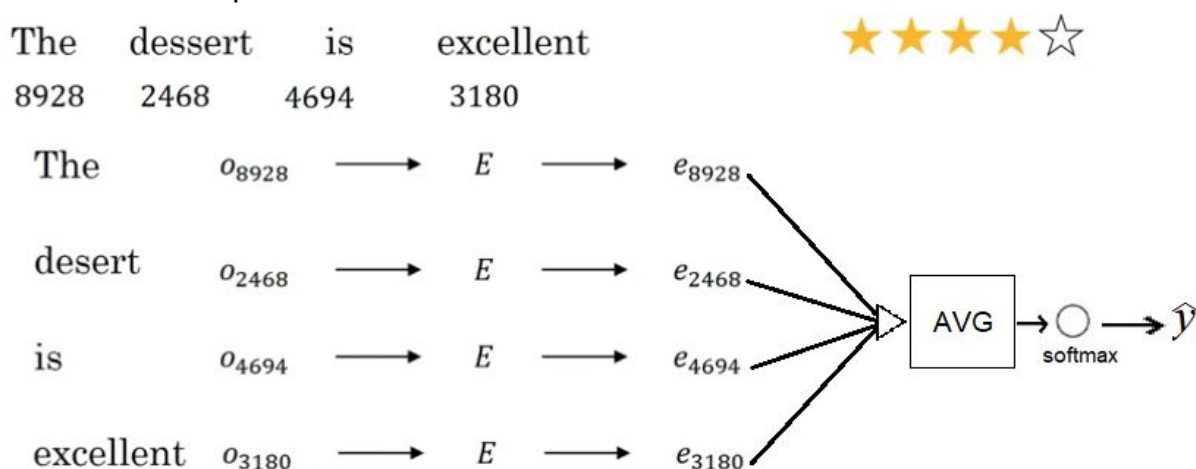
$-\log X_{ij} = \infty$. We'll use the convention that $0.\log(0) = 0$. The terms i and j are again playing the role of t and c . The b terms are biases.
This function allows you to learn meaningful word embeddings.

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 blocks in NLP and is used in many applications. One of the challenges of sentiment classification is you might not have a huge label training set for it, but with word embeddings you're able to build good sentiment classifiers with only modest-size label training sets. You can see some examples below:



Here is an example of how the sentiment would be classified:

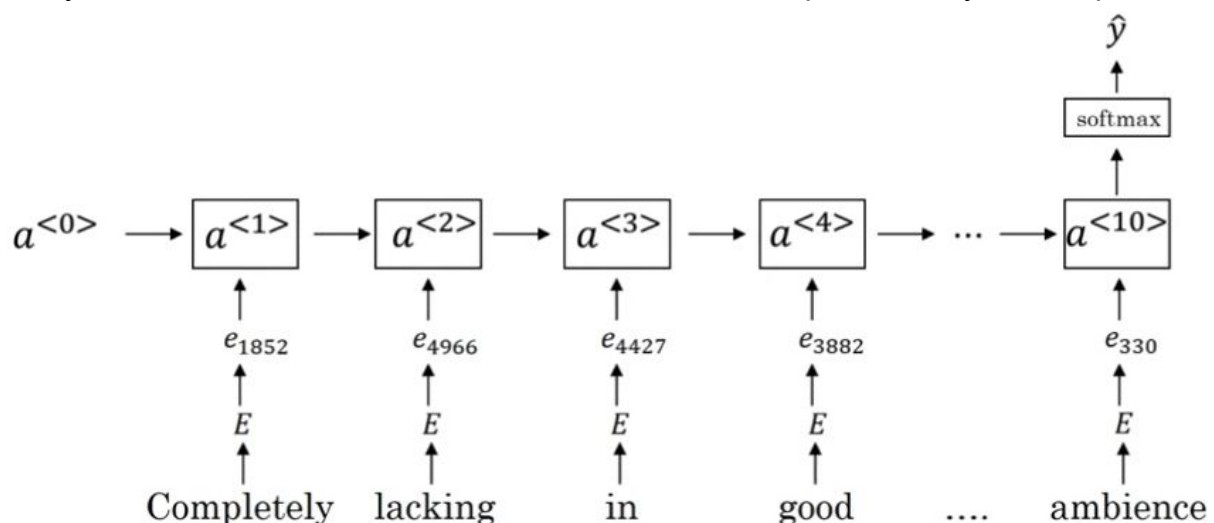


Using the one-hot vector of each one of these words and the embedding matrix with all the features, we'll compute the embedding vectors. We'll feed these vectors into a block that sum or average them and finally use a softmax layer to predict \hat{y} (you can train your embedding matrix with billions of words, if you have 300 features, your embedding vectors and the average/sum block will be 300 dimensional and the

softmax can output what are the probabilities of the five possible outcomes from one-star to five-stars).

However there is a problem with this algorithm. If it analyses the review “Completely lacking in good taste, good service, and good ambience.” the average would be higher and it will probably predict a good star, because there are a lot of “good” words and the algorithm doesn’t have the ability to understand the context of the sentence and see the word “lacking”.

To solve this problem you can use a RNN for sentiment classification. As usual, you multiply the one-hot vectors with the embedding matrix to get the embedding vectors, and then feed the embedding vectors into a RNN. The RNN is a many-to-one RNN, because it will have more than one input and only one output.



With an algorithm like this, it will be much better at taking word sequence into account and realizing that things like “lacking in good taste” is a negative review.

Debiasing word embeddings

Machine learning is increasingly trusted to help with extremely important decisions. So we 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. In this section you’ll learn some ideas for diminishing or eliminating these forms of bias in word embeddings. More details about this topic can be found in the paper “Bolukbasi et. al., 2016. Man is to computer programmer as woman is to homemaker? Debiasing word embeddings”. The title of the paper shows the problem of bias in word embeddings. The traditional algorithm would learn the following word embeddings:

Man:Woman as King:Queen

Man:Computer_Programmer as Woman:homemaker

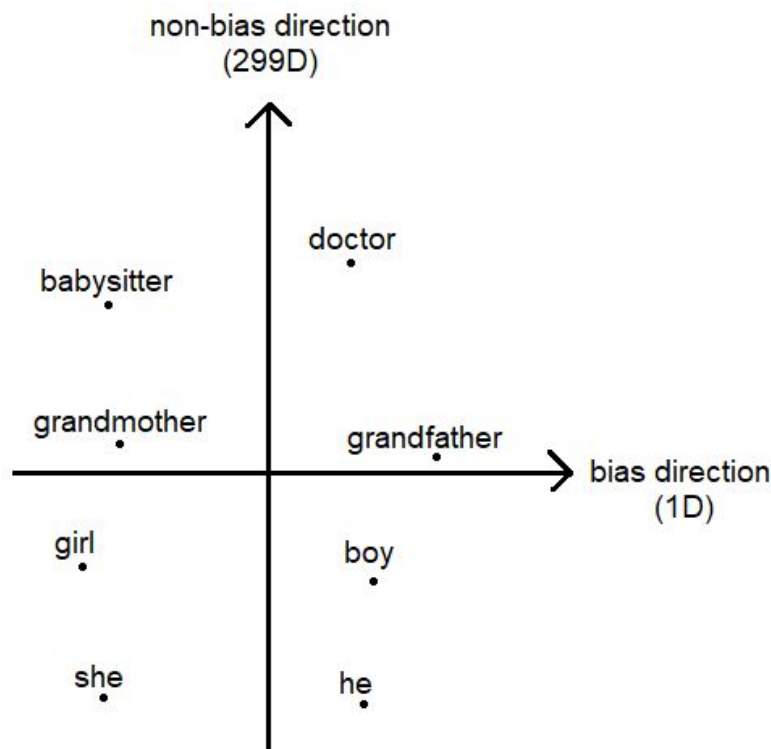
Father:Doctor as Mother:Nurse

As you can see, the second and third sentences are very unhealthy stereotypes. So removing the bias is a very important task that needs to be done because they can

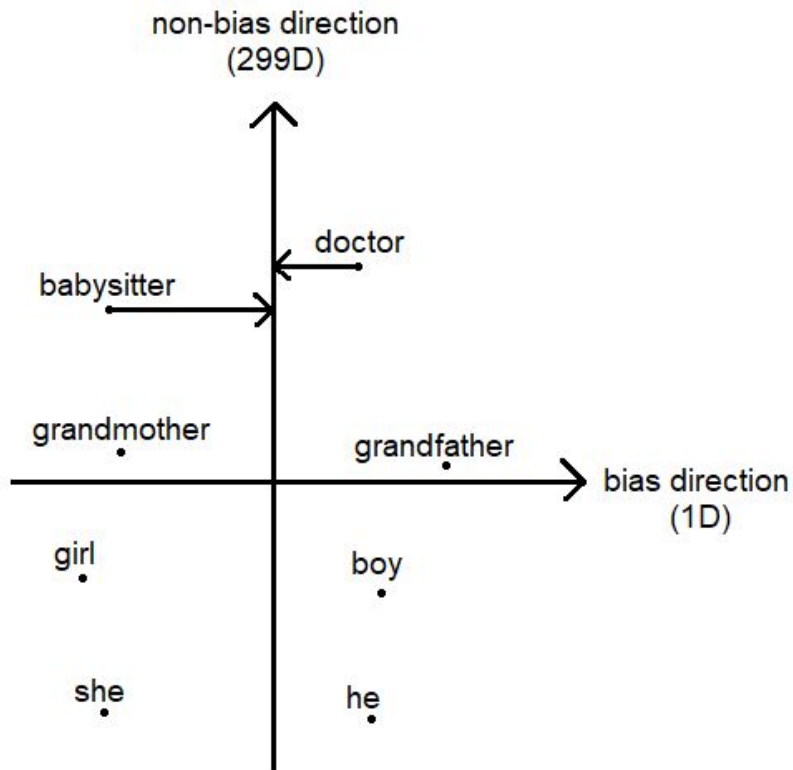
influence everything ranging from college admissions, to the way people find jobs, to loan applications, to in the criminal justice system and even sentencing guidelines. Word embeddings can reflect gender, ethnicity, age, sexual orientation, and other biases of the text used to train the model.

Let's learn now on how to address bias in word embeddings. The process you need to do is the following:

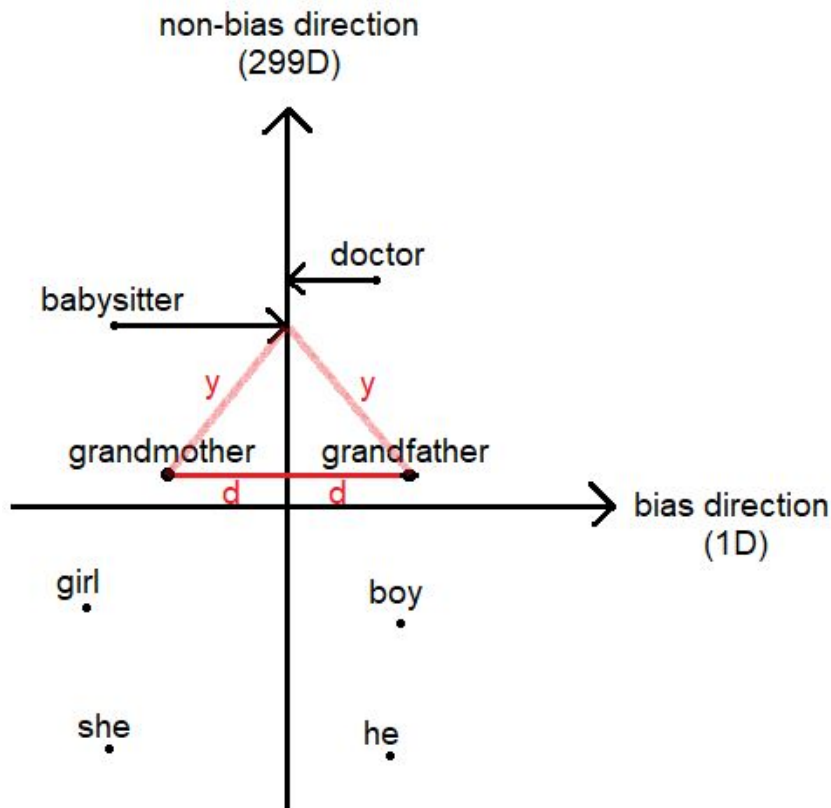
1. Identify the bias direction. If you have a gender bias, you can subtract some embedding vectors of different genders and compute the average between them. $e_{he} - e_{she} \mid e_{male} - e_{female} \dots$ then average them all together.



2. Neutralize: For every word that is not definitional, project to get rid of bias. In the more general case, you want words like doctor or babysitter to be ethnicity neutral and sexual orientation neutral, but in this case to simplify, we'll only use gender. We'll project words like "doctor" or "babysitter" that should be gender neutral in the non-bias axis, to remove the bias of these words.



3. Equalize pairs: which you might have pairs of words such as grandmother and grandfather, or girl and boy, where you want the only difference in their embedding to be the gender. In this example, the distance (similarity) between babysitter and grandmother is actually smaller than the distance between babysitter and grandfather. This may reinforces an unhealthy (or maybe undesirable) bias that grandmothers end up babysitting more than grandfathers. So in the equalization step, what we'd like to do is to make sure that words like grandmother and grandfather are both exactly the same distance from words that should be gender neutral. There are a few linear algebra steps for that, but what it will basically do is move grandmother and grandfather to a pair of points that are equidistant from the axis in the middle (the non-bias axis).



As you can see, both images are equidistant from the non-bias direction, and equidistant from babysitter.

The most part of the English words are not definitional, meaning that gender is not part of the definition and a relatively small subset of words should not be neutralized. A linear classifier can tell you what words to pass through the neutralization step to project out the bias direction, and the number of pairs you want to equalize.

These are the basic ideas of how to address the bias problem in word embeddings. For more details you can look at the paper mentioned in the beginning of the section.

Sequence models & Attention mechanism - Sequence to sequence architectures

Basic Models

In this section, you'll learn about sequence-to-sequence models, which are useful for everything from machine translation to speech recognition. You can learn more details about this process in the papers "Sutskever et. al., 2014. Sequence to sequence learning with neural networks." and "Cho et. al., 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation.". Let's say you want to input a French sentence like the one below:

“Jane visite l’Afrique en septembre.”

And you want to translate it to the English sentence:

“Jane is visiting Africa in September.”

As usual, let’s use the following notation to represent the French and English sentences:

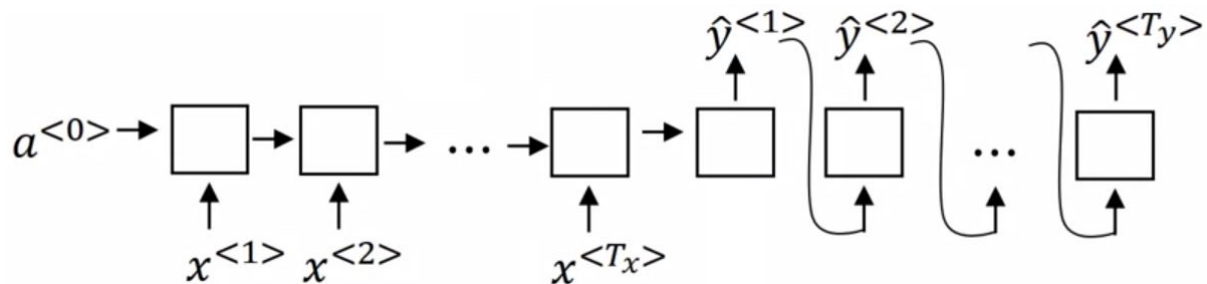
Jane visite l’Afrique en septembre.

$x^{<1>} \quad x^{<2>} \quad x^{<3>} \quad x^{<4>} \quad x^{<5>}$

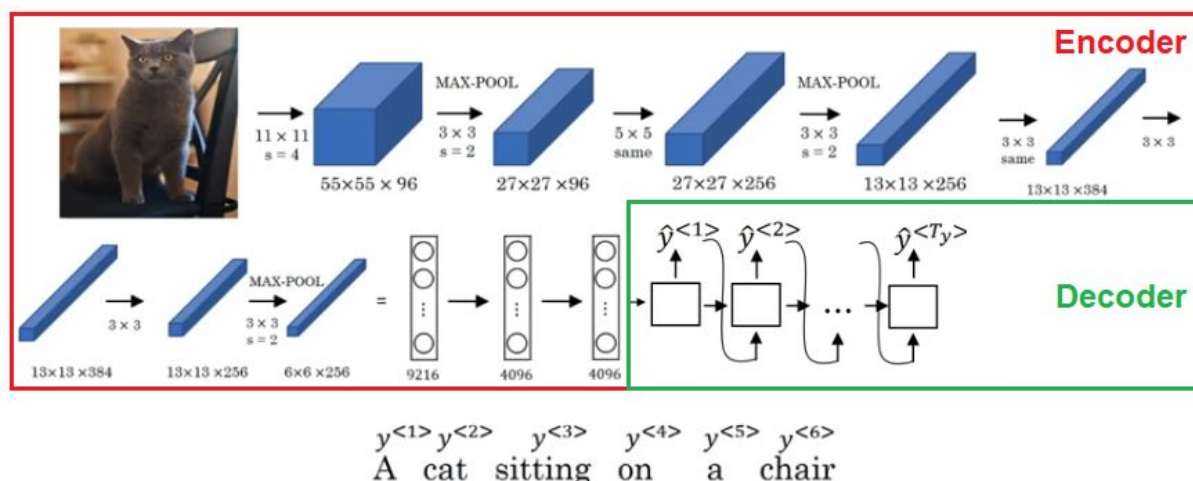
Jane is visiting Africa in September.

$y^{<1>} \quad y^{<2>} \quad y^{<3>} \quad y^{<4>} \quad y^{<5>} \quad y^{<6>}$

First, let’s have a network (which we’re going to call the encoder network) be built as a RNN, feed the input French words one word at a time. The input RNN is the encoder. After consuming the input sequence, the RNN then offers a vector that represents the input sentence. Finally, all these words from the encoder will be used by the decoder (that is also a RNN) to translate one word at a time.



Another application you can do is to make an image captioning system. You can remove the final softmax layer of a convolutional neural network, and all the layers that are left become the encoder. You can feed the last layer of the CNN (that will have features that represent the image) into a RNN network that has the job of generating the caption, one word at a time. For example:



For more details about this model, you can consult three different papers:

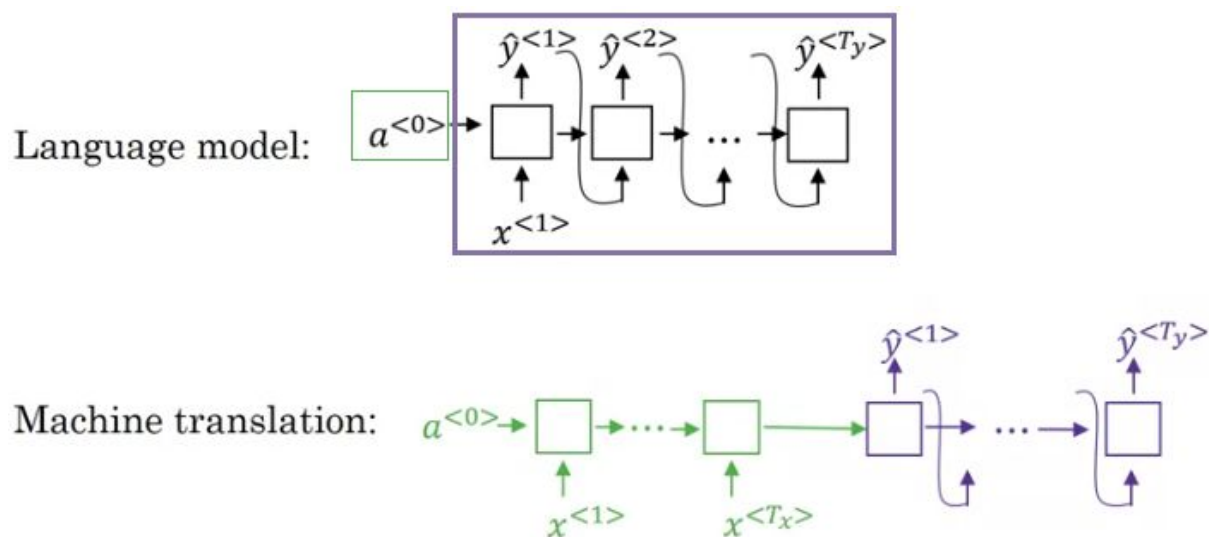
“Mao et. al., 2014. Deep captioning with multimodal recurrent neural networks.”

“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 sequence machine translation model and the language models that we’ve worked with in the first week of this course, as well as some differences. Let’s take a look at the image below to discuss more about that:



As you can see, the language model and the machine translation model have some similarities. We can think of the $a^{<0>}$ in the language model as being the encoder, and the rest of the RNN being the decoder. The machine translation model works like a conditional language model, because the probabilities of the translated words in the decoder depend on the input words in the encoder.

Language model: $P(y^{<1>}, \dots, y^{<T_y>})$

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

When you translate a sentence from one language to another, there are lots of possible sentences that are correct. The objective in this case is to choose the most likely sentence from all these sentences. One algorithm that allows us to choose the sentence is the beam search, that we are going to see more about in the next section.

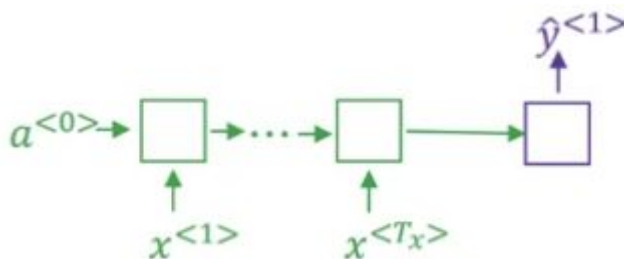
Jane visite l'Afrique en septembre. $P(y^{<1>}, \dots, y^{<T_y>} | x)$

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

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

Beam Search

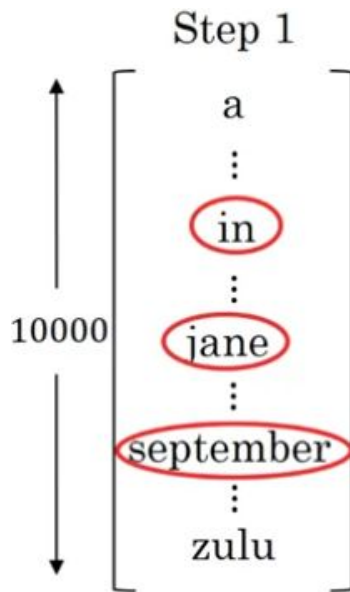
The beam search is an algorithm that tries to pick the most likely sentence. Let's use the sentence "Jane visite l'Afrique en septembre." and try to translate this word to English. The first thing the beam search has to do is try to pick the first words of the English translation that it's going to output. In the first step, let's use the following network:



With this network, we'll try to evaluate what is the probability of the first word, using softmax.

$$P(y^{<1>} | x)$$

Whereas greedy search will pick only the one most likely word and move on, beam search instead can consider multiple alternatives. Beam search has a parameter called B, which is called the beam width, and if you set the parameter B=3 for example, the beam search will consider three words at a time.



Having picked “in”, “jane” and “september” as the three most likely choices of the first word, what beam search will do now is for each of these three choices, what should be the second word. In terms of probability it will be expressed as:

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

Beam search will also analyse if one of the first words don't make sense anymore.

For example, imagine that these are the first and second words that are most likely to be the correct translation:

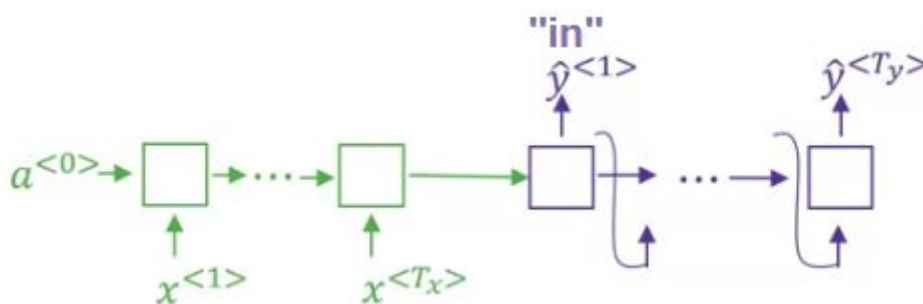
“in september ...”

“jane is ...”

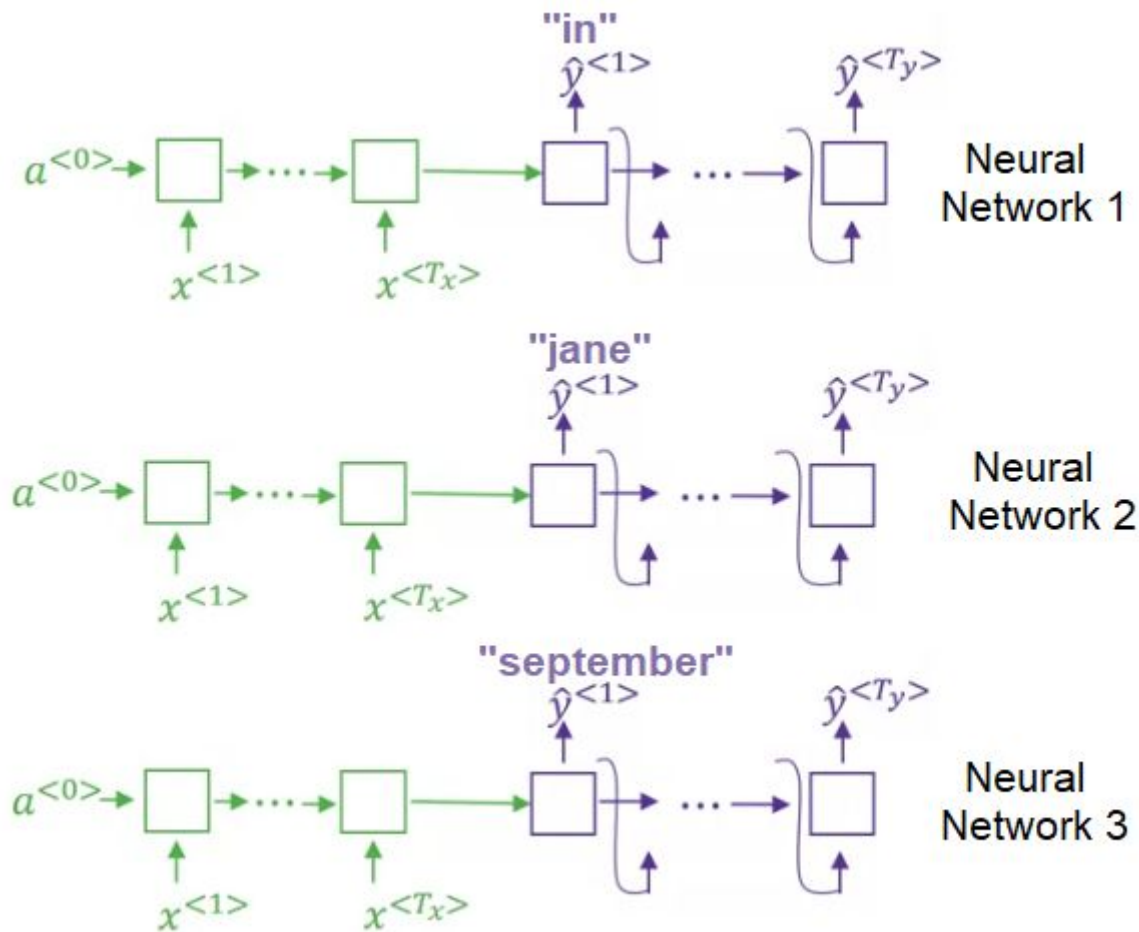
“jane visits ...”

In this case, the word “september” is eliminated as the possible first word, because it has a low probability of being the correct translation of the French sentence.

This process will repeat until all the sentences are translated.



For each chosen word, there will be 10000 possible next words to choose (considering that 10000 is the size of our vocabulary) and a copy of the neural network. After analysing all the words from the vocabulary using a softmax, we'll only pick three words as candidates to the next word ($B = 3$).



Notice that if we consider $B=1$, the beam search algorithm reduces to the greedy search algorithm.

Refinements to Beam Search

In this section you'll learn some little changes in the beam search algorithm that make it works even better.

Length normalization

The length normalization technique is used to normalize the values of the probabilities. The objective of beam search is to maximize the following:

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

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

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

However, if the probabilities are small numbers, these values can be very small and actually result in numerical underflow, meaning that it's too small for the floating part representation in our computers to store accurately. So instead of maximizing the product, we'll apply a log to the product, converting the product to a sum.

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

This process should get rid of the numerical underflow problem and give the same results as before.

There is another problem with these equations: the algorithm will prefer smaller sentences instead of bigger sentences, because the multiplication of a higher quantity of numbers between 0 and 1 (or the sum of logs between 0 and 1) results in a small number. The algorithm should prefer smaller, but not always likely sentences to output because of this problem. So instead of maximizing the object functions, we'll take the average:

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

This equation significantly reduces the penalty for outputting longer sentences.

Obs: you can actually use different heuristics to try to optimize the results. You can use the following function, that uses a hyperparameter α (that can be tuned) to try to use a middle term between full normalization ($\alpha = 1$) and no normalization ($\alpha = 0$).

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

Each neural network will output a candidate to the most likely sentence and the goal is to choose the one that performs better in the object function above.

Beam width B

The larger the value of B is, the more possibilities you're considering and you'll probably find better sentences. But the larger B is, the more computationally expensive your algorithm is, because you are also keeping a lot more possibilities around. Here are some pros and cons of setting B to be very large versus very small:

Large B:

- Better results but slower.

Small B:

- Worse results, but faster.

In production systems, it's not uncommon to see a beam with a value of 10. A beam width of 100 would be considered very large for a production system, depending on the application. For research systems, where people want to squeeze out every last drop of performance in order to publish the paper with the best possible results, it's not uncommon to see people using beam widths of 1000 or 3000. So the idea is to

try different values of B as you work through your application, to get a good balance between results and performance.

Obs: Unlike exact search algorithms like BFS (Breadth First Search) or DFS (Depth First Search), Beam Search runs faster but is not guaranteed to find exact maximum for $\arg \max_y P(y | x)$.

Error analysis in beam search

Beam Search is an approximate search algorithm, also called a heuristic search algorithm. It doesn't always output the most likely sentence. What if beam search makes a mistake? In this section 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 or whether it might be your RNN model that is causing problems. So this section should help you to decide where to spend time on when some problem occurs. Let's start with an example:

Sentence: "Jane visite l'Afrique en septembre."

Human translation: "Jane visits Africa in September." (y^*)

Algorithm: "Jane visited Africa last September." (\hat{y})

As you can see, the algorithm doesn't output a good translation. There are two components in your model, the RNN and the beam search. Let's try to figure out which one of these components is causing problems. What you can do to check if your RNN is performing correctly, is to compute the probabilities of the sentences y^* and \hat{y} to see what sentence gives the bigger probability. There is going to be two cases:

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

Beam search choose \hat{y} , but y^* attains higher $P(y | x)$. The conclusion of this result is that beam search is at fault.

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

y^* is a better translation than \hat{y} . But RNN predicted $P(y^* | x) < P(\hat{y} | x)$. The conclusion of this result is that the RNN model is at fault.

What you can do to make a better analysis is to create a table:

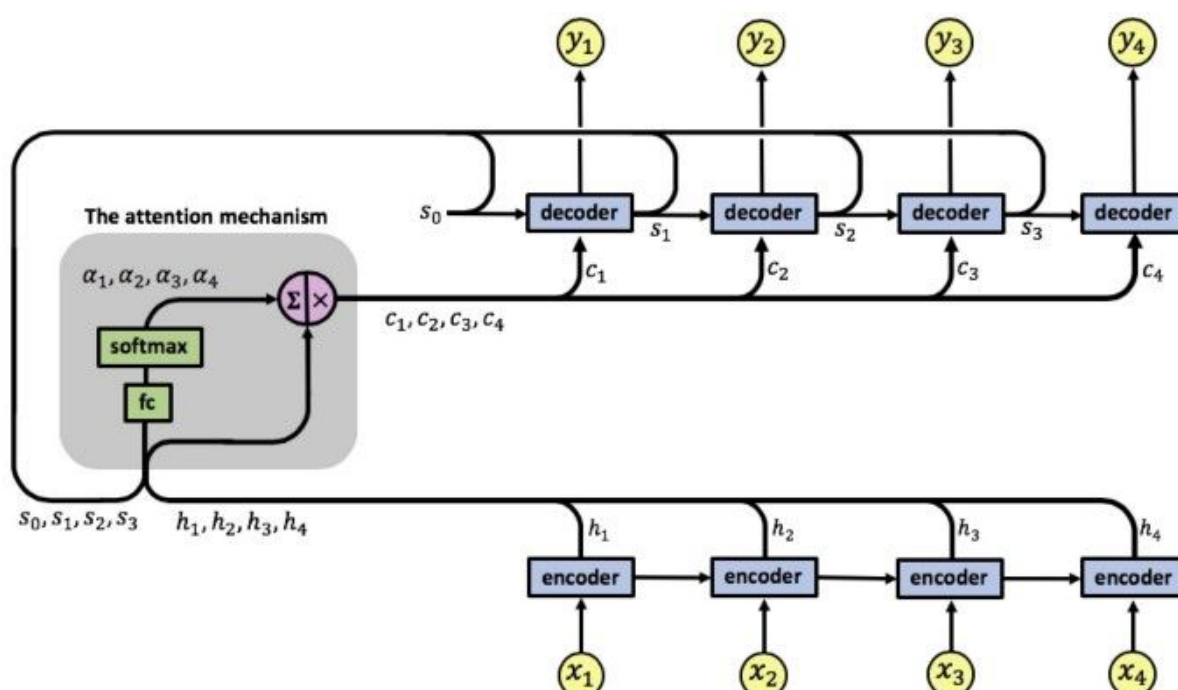
Human	Algorithm	$P(y^* x)$	$P(\hat{y} x)$	At fault?
Jane visits Africa in September.	Jane visited Africa last September.	$2 * 10^{-10}$	$1 * 10^{-10}$	B
...	R
...

This table will help you to better visualize which component is causing some troubles in your model and figure out what fraction of errors are “due to” beam search vs. RNN model.

Attention Models

In this course, you’ve been using an encoder-decoder architecture for machine translation, where one RNN reads a sentence and then a different one outputs a translated sentence. There’s a modification to this called the Attention Model, that makes all this work much better. More details about the Attention Model can be seen in the paper “Bahdanau et. al., 2014. Neural machine translation by jointly learning to align and translate.”.

The encoder-decoder architecture works well for short sentences, but with sentences longer than 30 or 40 words, the performance usually comes down. The Attention Model is a solution to this problem, because it allows the translation of bigger sentences just like humans do, translating parts of the sentence at a time. An attention RNN looks like this:



Our attention model example has a single layer RNN encoder, with 4-time steps. We denote the encoder’s input vectors by x_1, x_2, x_3, x_4 and the output vectors by h_1, h_2, h_3, h_4 .

The attention mechanism is located between the encoder and the decoder, its input is composed of the encoder’s output vectors h_1, h_2, h_3, h_4 and the states of the decoder s_0, s_1, s_2, s_3 (the states work just like the activations). The attention’s output is a sequence of vectors called context vectors denoted by c_1, c_2, c_3, c_4 .

The context vectors enable the decoder to focus on certain parts of the input when predicting its output. Each context vector is a weighted sum of the encoder's output vector h_1, h_2, h_3, h_4 . Each vector h_i contains information about the whole input sequence, with a strong focus on the parts surrounding the i -th vector of the input sequence. The vectors h_1, h_2, h_3, h_4 are scaled by weights α_{ij} capturing the degree of relevance of input x_j to output at time i, y_i .

The context vectors c_1, c_2, c_3, c_4 are given by:

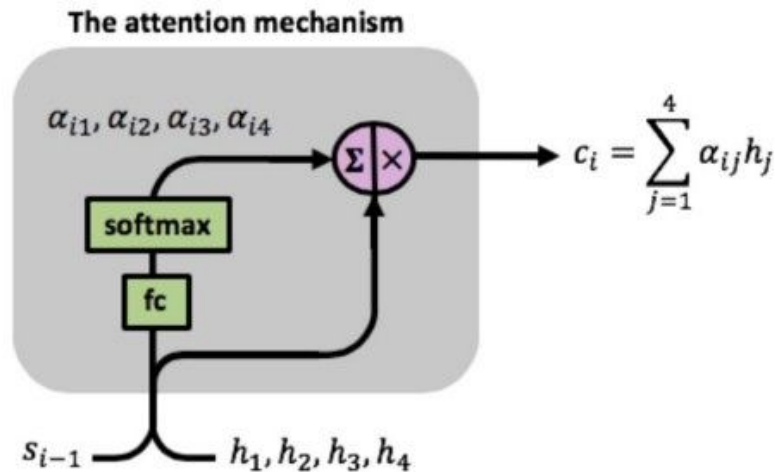
$$c_i = \sum_{j=1}^4 \alpha_{ij} h_j$$

The attention weights are learned using an additional fully-connected shallow network, denoted by fc , this is where the s_0, s_1, s_2, s_3 part of the attention mechanism's input comes into play. The computation of the attention weights is given by:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^4 \exp(e_{ik})}$$

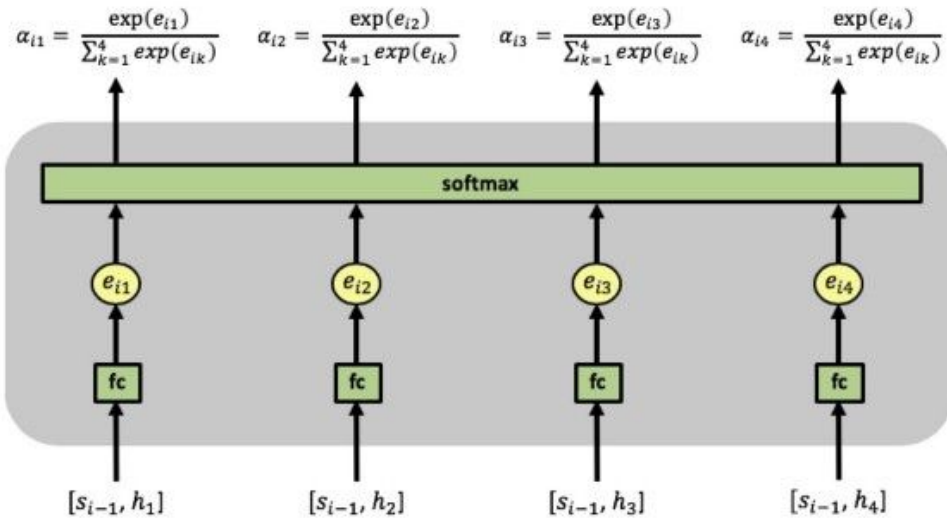
$$\text{where } e_{ij} = \text{fc}(s_{i-1}, h_j)$$

The attention weights are learned using the attention fully-connected network and a softmax function:



As can be seen in the image above, the fully-connected network receives the concatenation of vectors $[s_{i-1}, h_i]$ as input at time step i . The network has a single fully-connected layer, the outputs of the layer, denoted by e_{ij} , are passed through a softmax function computing the attention weights, which lie in $[0, 1]$.

Notice that we are using the same fully-connected network for all the concatenated pairs $[s_{i-1}, h_1], [s_{i-1}, h_2], [s_{i-1}, h_3], [s_{i-1}, h_4]$, meaning there is a single network learning the attention weights.



The attention weights α_{ij} reflect the importance of h_j with respect to the previous hidden state s_{i-1} in deciding the next state s_i and generating y_i . A large α_{ij} attention weight causes the RNN to focus on input x_j (represented by the encoder's output h_j), when predicting the output y_i .

The fc network is trained along with the encoder and decoder using backpropagation, the RNN's prediction error terms are backpropagated backward through the decoder, then through the fc attention network and from there the encoder.

A RNN with 4 input time steps and 4 output time steps will have the following weight matrices fine-tuned during the training process. Note the dimensions 4x4 of the attention matrix, connecting between every input to every output:

RNN encoder weights matrix W_e

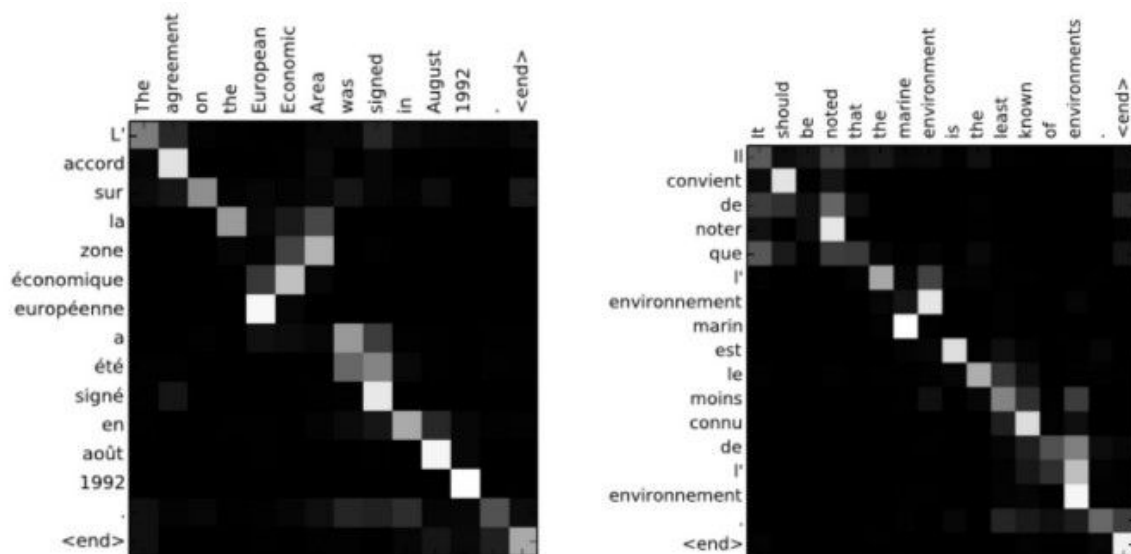
RNN decoder weights matrix W_d

RNN attention weights matrix $\alpha_{4 \times 4}$

fc weights matrix W_a

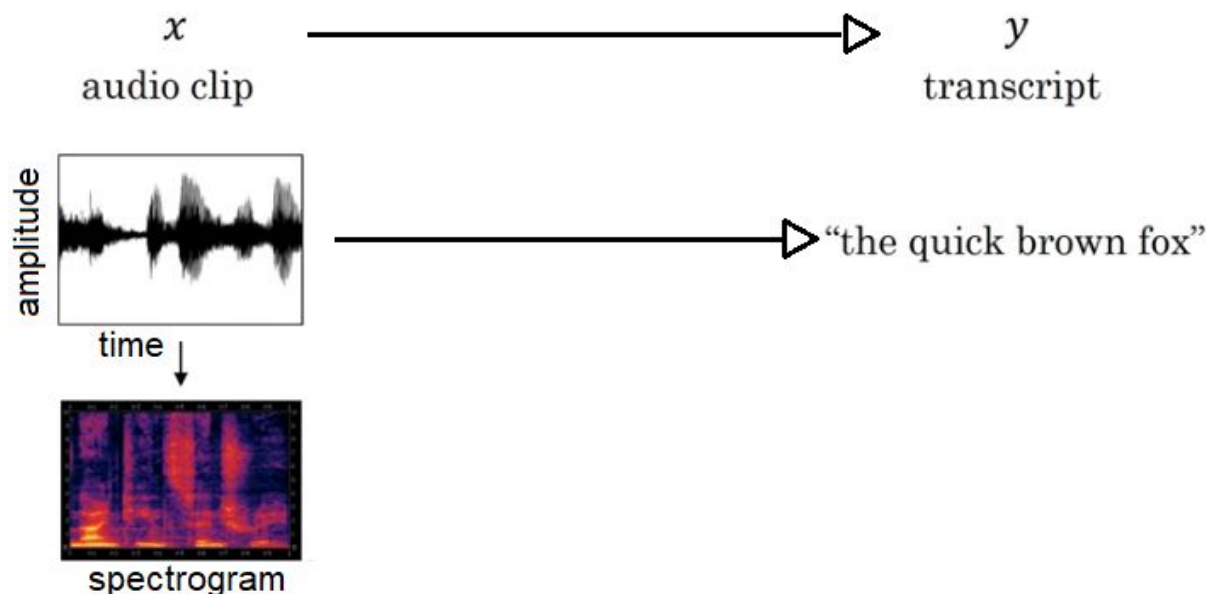
This mechanism enables the decoder to decide which parts of the input sequence to pay attention to. By letting the decoder have an attention mechanism, we relieve the encoder from having to encode all information in the input sequence into a single vector. The information can be spread throughout the sequence h_1, h_2, h_3, h_4 which can be selectively retrieved by the decoder.

An example of the attention model in practice can be seen in the picture below, where each pixel shows the weight α_{ij} of the j -th source word and the i -th target word, in grayscale (0: black, 1: white).



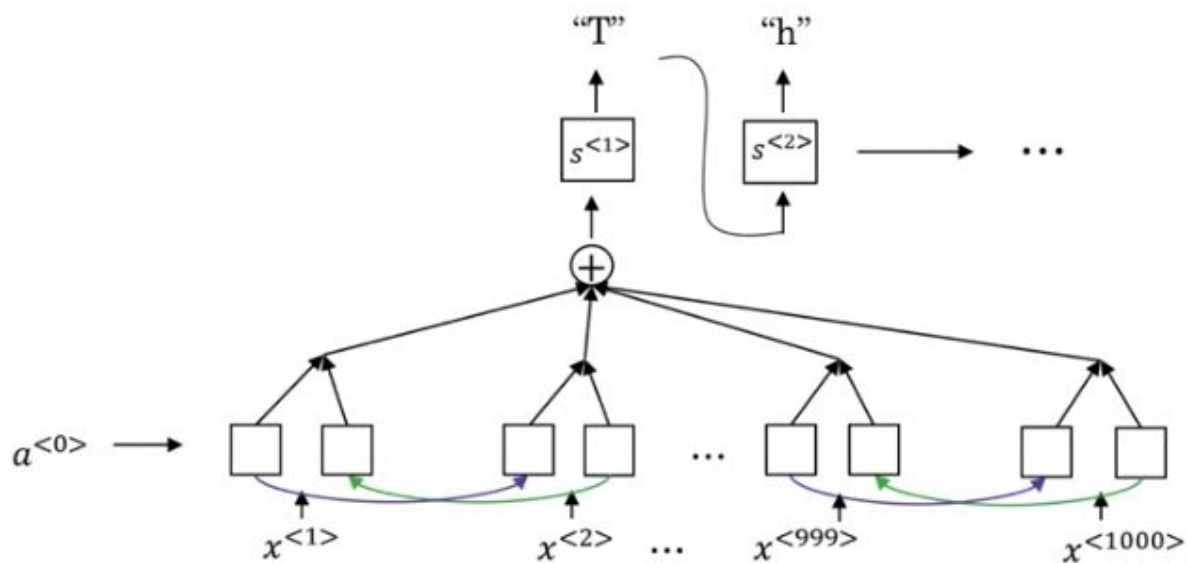
Speech Recognition

One of the most exciting developments where sequence-to-sequence models are used is on the speech recognition applications. Let's see what is a speech recognition problem:



From an input clip, the objective is to transcribe an output y . The audio clips are usually represented as time series, with the time in the horizontal axis and the respective amplitude of the audio in the vertical axis. To optimize the transcription, there is a common pre-processing step for audio data that is to run your raw audio clip and generate a spectrogram. The time is represented in the horizontal axis, the frequency is represented in the vertical axis, and the intensity of different colors shows the amount of energy.

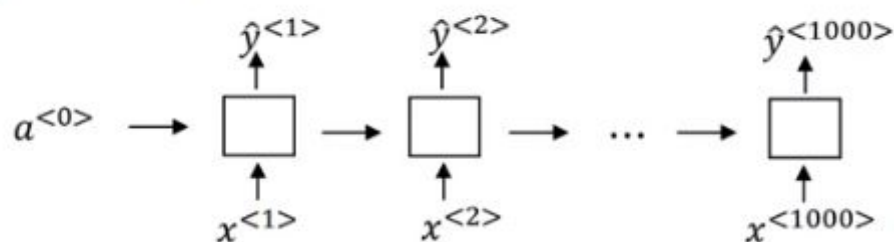
An example of how an attention model can be used to perform speech recognition can be seen below:



One other method that works well is to use the CTC cost for speech recognition. CTC stands for “Connectionist temporal classification”, and it’s due the paper “Graves et. al., 2006. Connectionist Temporal Classification: Labeling unsegmented sequence data with recurrent neural networks.”. This is how it works:

Let’s work with a many-to-many RNN.

“the quick brown fox”



In audio speech recognition, usually the amount of input data is smaller than the amount of output data. If we have a audio sequence of 10 seconds with 100Hz of frequency, the input is going to have 1000 elements, while the output can have just a few elements (each character = 1 output). The output, considering the sentence “the quick brown fox”, using a CTC is going to be like this:

t t t _ h _ e e e _ _ _ _ q q q _ _ _ ...

The basic rule of the CTC cost is to collapse repeated characters not separated by “blank” (the “blank” is represented by “_”). So the earlier output is going to be like this:

the q...

This process allows the neural network to have multiple outputs, making the number of outputs equals to the number of inputs. The sentence “the quick brown fox” has 19

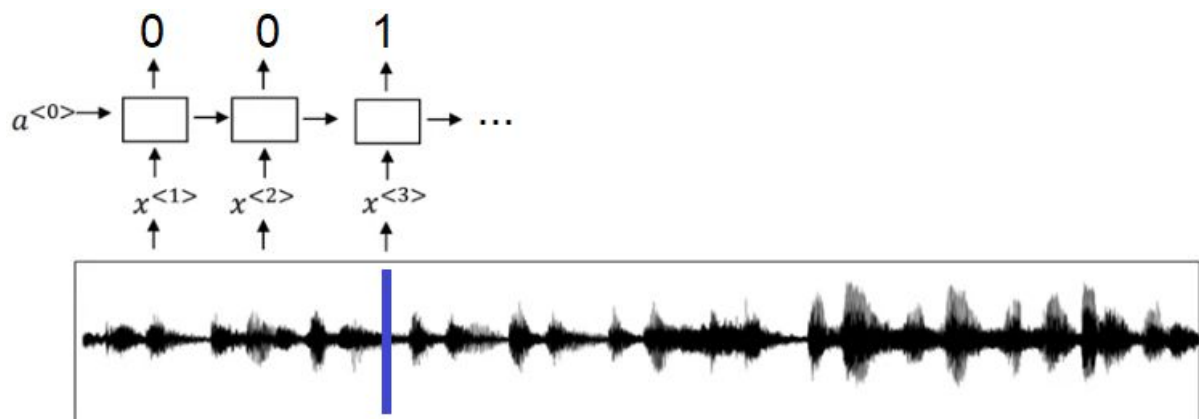
characters, and using the CTC cost allows us to add extra outputs in the neural network.

Trigger Word Detection

You've learned so much about deep learning and sequence models, that we can actually descobre a trigger word system quite simply. Examples of trigger word systems include:



The literature on triggered detection algorithms is still evolving, so there isn't wide consensus yet on what's the best algorithm for trigger word detection. One thing you can do is the following:



The idea is to train your RNN to output a value of 1 when the trigger word is recognized. This type of labeling scheme will work quite well, but it has one slight disadvantage that it creates a very imbalanced training set, so we have a lot more zeros than we want. One little “hack” we can do is to output multiple “1” each time the trigger word is recognized (like 5 or 6 “1”) instead of just a single “1”. With the trigger word recognized, you can link this network with other networks to recognize other sentences, such as:

“Turn off the lights”

“What time is it?”

“How is the weather?”

This is just a basic idea of how trigger word detection works. There are more sophisticated algorithms to perform this task and you can search more about them on the internet.

END OF THE FIFTH AND FINAL COURSE