Sequence Models

Sequence models are the machine learning models that input or output sequences of data. Sequential data includes text streams, audio clips, video clips, time-series data and etc. Recurrent Neural Networks (RNNs) is a popular algorithm used in sequence models

Recurrent Neural Networks

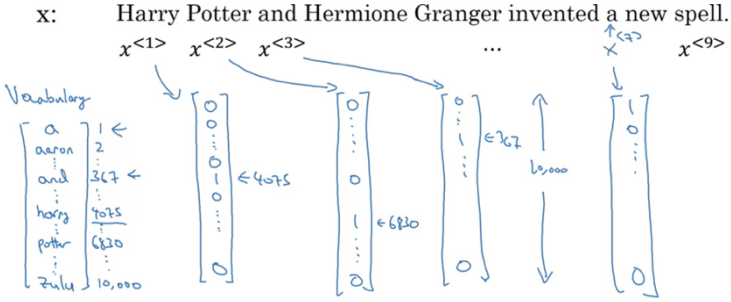
Notation

$$x^{(i)<t>} := t^{th} element\ in\ i^{th} example\ of\ input$$

$$y^{(i)<t>} := t^{th} element\ in\ i^{th} example\ of\ output$$

$$T_x^i = the\ length\ of\ i^{th} input\ example$$

Representing words – Vocabulary/ Dictionary

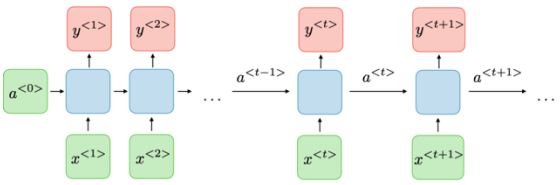


With a large vocabulary, we represent each word in input examples as one-hot encoding. And if there is some words not in the vocab, we represent it as <UNK>

Why not a standard network?

- Input and output can be different lengths in different examples
- Doesn't share features learned across different positions of text

What is Recurrent Neural Network?



Recurrent neural networks, also known as RNNs, are a class of neural networks that allow previous outputs to be used as inputs while having hidden states.

Parameters W_{ax} , W_{ya}, W_{aa} , are the same from X1 to the hidden layers, $a^{<0>}$ is a vector of zeros

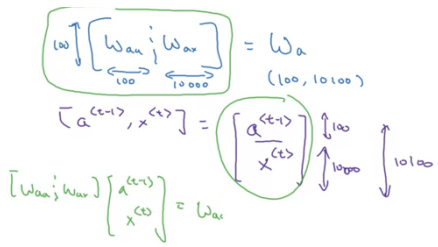
For each timestep, the activation , $a^{<t>}$ and the output $y^{<t>}$ are expressed as follows

$$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a) \rightarrow g_1(W_a\,[a^{<t-1>}, x^{<t>}] + b_a)$$

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

To understand the notation, W_{ax} means the input it will multiply with is a x(right side), the output it will be is an activation(left side), similarly, for W_{ya}, the input is an activation, the output is a y

To simplify the equation:

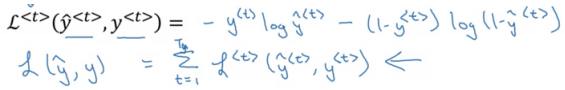


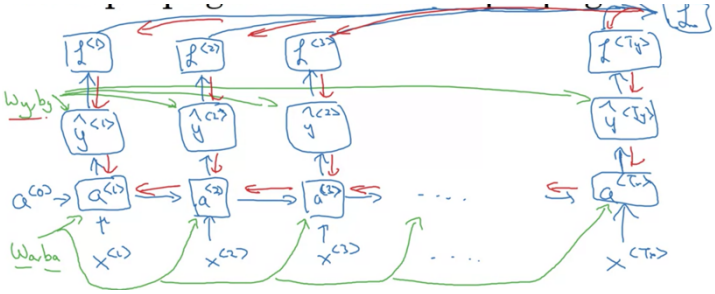
W_{aa} and W_{ax} combined as W_a, $a^{<t-1>}, x^{<t>}$ stack together
Similar as W_{ya} changes to W_y, so that we only focus on the output when we write the notation.

Advantages	Drawbacks
• Possibility of processing input of any length • Model size not increasing with size of input • Computation takes into account historical information • Weights are shared across time	• Computation being slow • Difficulty of accessing information from a long time ago • Cannot consider any future input for the current state

Backpropagation Through Time

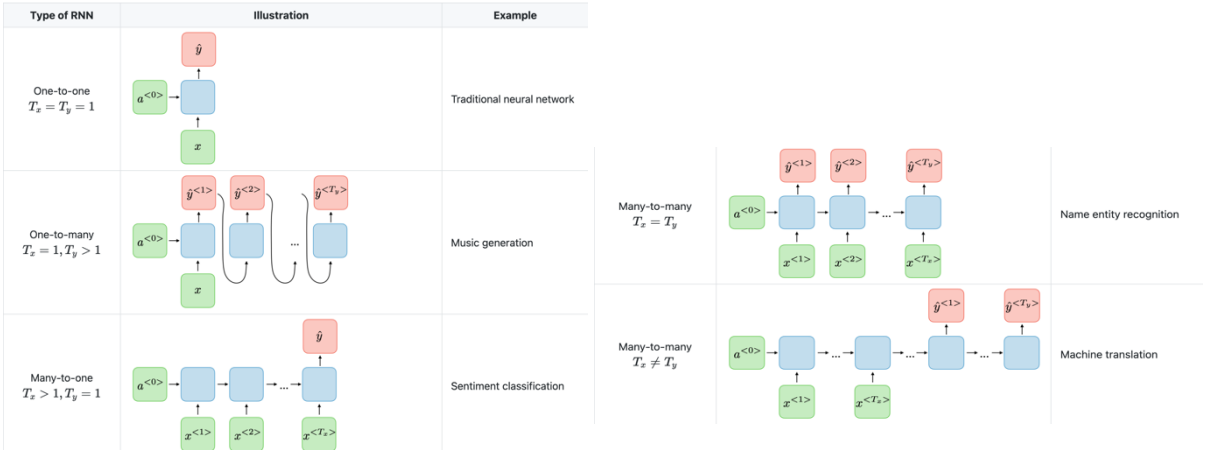
Loss and Cost function





Since we need to go from a<Tx> to a^0, so it's called backprop through time

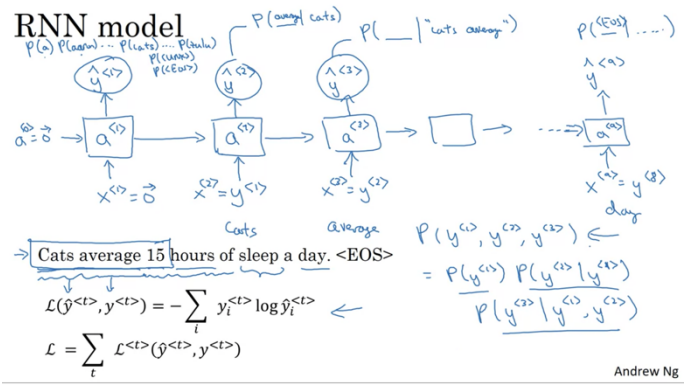
Different Types of RNN



Language Model and Sequence Generation

What Language Model does is given any sentence, it will tell you what the probability of that sentence will be

Given a sentence input, it will create a dictionary for the words (**tokenize**, similar as OHE), put a token as <EOS> to specify the end point of a sentence, punctuations can be added to the dictionary as well



For each input, it takes activation and the output of the previous layer. For example, the input for y_hat2 is y_1, which is the output of time step1, and a_2, which is the activation of time step2
For each time step, we will output the probability of each word in the dictionary, and get the highest probability word as our output, for example, the first time step output is cat. Then the second time step take cat as part of its input, so its output would be P(word in the dic|cats), the third time step output would be P(word in the dic | "cat average")
Then we will train the model with the loss function and aggregate loss functions for all time steps to get the cost function. This model can help us predict the remaining sentence with some words input. For example, when we have "cats average 15" as input, it can help us predict the rest of the sentence
However, when learning, the output of an RNN is a probability distribution instead of one word.
When generating text we choose only **one** of the words ourselves given the probabilities and feed that back into the network. This is called **sampling**.
To begin character-level sampling:

- Input a "dummy" vector of zeros as a default input
- Run one step of forward propagation to get $a\langle 1\rangle$ (your first character) and $\hat{y}\langle 1\rangle$ (probability distribution for the following character)
- When sampling, avoid generating the same result each time given the starting letter by using np.random.choice

You can think of the recurrent neural network as the repeated use of a single cell. First, you'll implement the computations for a single time step. The following figure describes the operations for a single time step of an RNN cell

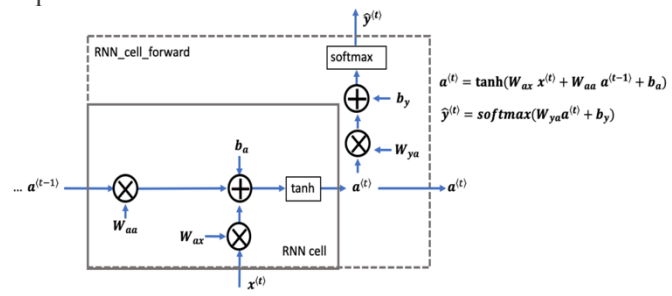


figure as the outer box with dashed lines

Note that an RNN cell outputs the hidden state $a\langle t\rangle$a⟨t⟩.

- RNN cell is shown in the figure as the inner box with solid lines
- The function that you'll implement, rnn_cell_forward, also calculates the prediction $\hat{y}\langle t\rangle$y^⟨t⟩
- RNN_cell_forward is shown in the

RNN Forward Pass Implementation¶
A recurrent neural network (RNN) is a repetition of the RNN cell that you've just built.
If your input sequence of data is 10 time steps long, then you will re-use the RNN cell 10 times
Each cell takes two inputs at each time step:
$a\langle t-1\rangle$: The hidden state from the previous cell
$x\langle t\rangle$: The current time step's input data
It has two outputs at each time step:
A hidden state ($a\langle t\rangle$), A prediction ($y\langle t\rangle$)
The weights and biases (Waa,ba,Wax,bx)(Waa,ba,Wax,bx) are re-used each time step
BACKWORD Propagation

$$a^{\langle t \rangle} = \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b_a)$$

$$\frac{\partial \tanh(x)}{\partial x} = 1 - \tanh^2(x)$$

$$dtanh = da_{next} * (1 - \tanh^2(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b_a))$$

$$dW_{ax} = dtanh \cdot x^{\langle t \rangle T}$$

$$dW_{aa} = dtanh \cdot a^{\langle t-1 \rangle T}$$

$$db_a = \sum_{batch} dtanh$$

$$dx^{\langle t \rangle} = W_{ax}{}^T \cdot dtanh$$

$$da_{prev} = W_{aa}{}^T \cdot dtanh$$

Character Level language Model

Instead of only getting English words in our dictionary, we can build a character-level dictionary
And if you build a character level language model rather than a word level language model, then your sequence y1, y2, y3, would be the individual characters in your training data, rather than the individual words in your training data. So instead of y1 = cats, we will have y1 = c, y2 = a etc..
Pros: Don't need to worry about UNK words. For those rare words in out input, we don't need to worry about assigning a zero prob to it, character-level dictionary can learn any words as long as it has characters.
Cons: will have a much longer sequence, computational expensive

Vanishing Gradient in RNN

The basic RNN we've seen so far is not very good at capturing very long-term dependencies.
For example, when we have a sentence input "The cats, which already ate … , were at home", the "Cats" at the very beginning of the sentence may has a hard time affect "were", which at the end of the sentence. So that, when we do backpropagation, the calculation of the end time step, may not be able to affect the gradients in the beginning of the time step.
It's difficult for the output here to be strongly influenced by an input that was very early in the sequence. This is because whatever the output is, whether this got it right, this got it wrong, it's just very difficult for the error to backpropagate all the way to the beginning of the sequence, and therefore to modify how the neural network is doing computations earlier in the sequence.
It turns out that **exploding gradients** are easier to spot because the parameter has just blow up. You might often see NaNs, not a numbers, meaning results of a numerical overflow in your neural network computation. If you do see exploding gradients, one solution to that is apply gradients clipping.
All that means is, look at your gradient vectors, and if it is bigger than some threshold, re-scale some of your gradient vectors so that it's not too big, so that is clipped according to some maximum value.

Solution to Vanishing Gradients – Gated Recurrent Unit (GRU)

To solve the vanishing gradient problem of a standard RNN, GRU uses **update gate** and **reset gate,** deciding which and what information should be passed to the output. With these two trained parameters, we can save information for a long term, like from the starting time step to the end time step. This is called **the Gated Hidden State,** same as the output of activation function

1. Update gate

update gate would allow us to control how much of the new state is just a copy of the old state.
We start with calculating the update gate (gamma) $\Gamma_u = \sigma(W_u[c^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_u$, u here denotes for "update", we use sigmoid function to keep the value close to 0 or 1
Here $c^{\langle t-1 \rangle}$is actually the **memory cell** , usually the same as $a^{\langle t-1 \rangle}$, memory cell may contain information like plural or singular, and update gate has the ability to know when to update/ change the value of memory cell. For example, in the sentence "The cat, which ate …, was full", the memory cell change to 1 at "cat" indicating singular, till "was", and the update gate will know it's used, and we no longer need this information, and will forget it.

2. Calculate the candidate to replace memory cell

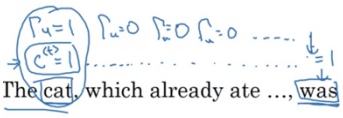
$\tilde{c}^{\langle t \rangle}$ would be a candidate to replace $c^{\langle t \rangle}$ and $\tilde{c}^{\langle t \rangle} = \tanh(W_c[\Gamma_r * c^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_c)$

Γ_r denotes how relevant $c^{<t-1>}$ is to update $\tilde{c}^{<t>}$

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

3. Calculate memory cell value

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



so in this example, we set the $\Gamma_u = 1$ at cat, so we update c there, and keep $\Gamma_u = 0$ all the way till "was", basically saying "don't update, remember the previous c", so that in the equation $c^{<t>} = c^{<t-1>}$, which uses the previous value of memory cell.

Long Short Term Memory(LSTM)

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

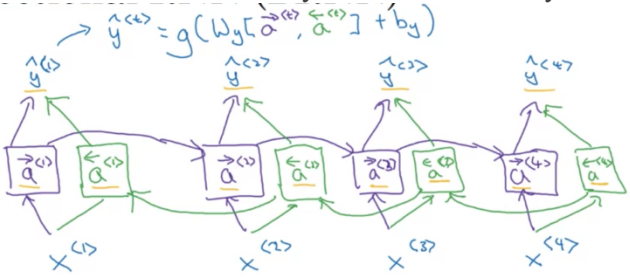
In LSTM, it's not possible for $c^{<t>} = a^{<t>}$ anymore. Instead of only having update and relevant gates, LSTM will have **update gate, forget gate** and **output gate.**

The output would be:

$$\mathbf{y}^{\langle t \rangle}_{pred} = \text{softmax}(\mathbf{W}_y \mathbf{a}^{\langle t \rangle} + \mathbf{b}_y)$$

Bidirectional RNN

No matter for LSTM, GRU, or basic RNN, they all are forward process, but sometimes it's not enough to predict the following words only based on the previous words. For example, with only knowing "Teddy", we are not sure if it's "Teddy bear" or it's "Teddy Roosevelt"



BRNN allows us to take both previous and future output as input to make prediction

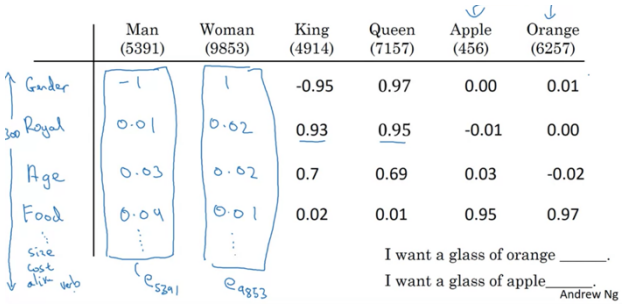
Deep Neural Network

$a^{[l]<t>}$ means activation for layer l and time step t

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

Word Embeddings

Word representation: store words in dictionary and use One-Hot encoding for representation
Featurized representation: word embedding-> extract feature from each word -> for each feature, apply probabilities to each word that it could belong to certain feature.
one-hot vectors don't do a good job of capturing the level of similarity between words. This is because every one-hot vector has the same Euclidean distance from any other one-hot vector.
Embedding vectors, such as GloVe vectors, provide much more useful information about the meaning of individual words.



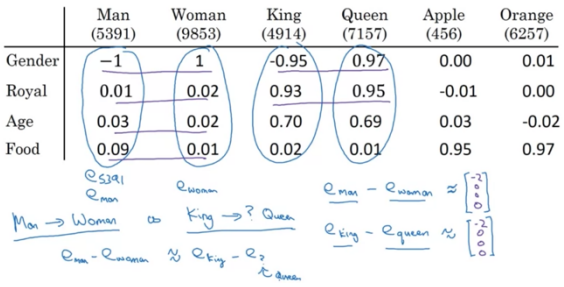
take the pic as an example, e_{5391} represent the feature probability for "Man", and Man is the 5391 word in the word dictionary.
Why do we need this? See if we want to predict the words after "I want a glass of apple _" we have "I want a glass of orange juice" in the training, and since orange and apple have very similar feature probability vectors, the words after them could be the same.

t-SNE takes a high-dimensional data and represent it in lower dimension without lose too much information.

Word Embedding allows transfer learning:

1. Learn word embeddings from large text corpus (or download online)
2. Transfer embedding to new task with **smaller** training set (if we have a really large dataset, the transfer learning might not work, it might requires specific text corpus for the specific task, such as machine translation)
3. Optional, if the training is a little large, we can continue to finetune the word embedding with new data

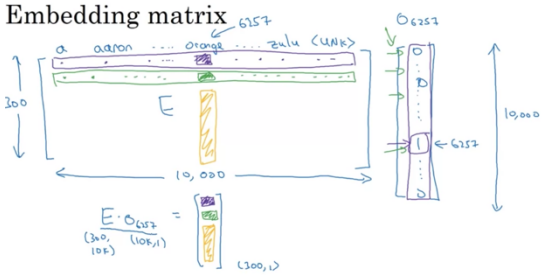
Properties of Word Embedding



word embedding can help with identifying analogies. For example, here $e_{man} - e_{woman} \approx e_{king} - e_{queen}$, so if we want to find $e_{man} - e_{woman} \approx e_{king} - e_?$, we can calculate the property difference between king and other words.

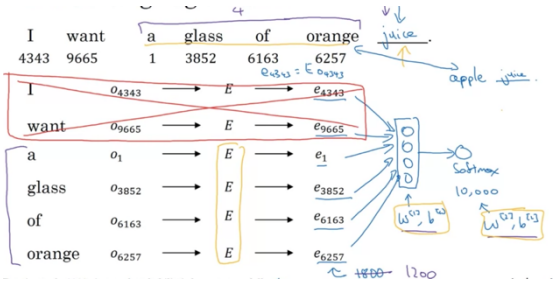
$$\arg\max w \ \ similarity(e_w, e_{king} - e_{man} + e_{woman})$$

$$Cosine\ Similarity = \frac{u^T v}{\|u\|\|v\|}$$



For matrix E (feature probability), the 10,000 cols represent words in our dictionary, the 300 rows represent feature probability for each word. If we have a word "orange" as input, O_{6257} represents the One hot vector with only 1 at 6257th position, because orange is the 6257th word in the dictionary. If we multiply E with O, we will get the feature vector for "Orange"

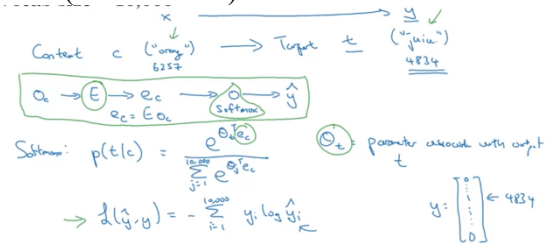
Natural Language Model



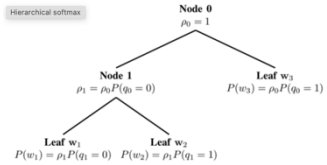
with a sentence as input, we use E as parameter, as we discussed above, to generate vector e for each word and stacking them together. So each word has a feature probability vector with 300 dimensions, this examples hidden layer would have 300*6 =1800 dimensions. Then we can use gradient descent to update w and b to optimize our learning. Also, we can design a window, say, we only input the previous 4 words before our target word. So the dimension would be 4*300 = 1200. The number of words taking into account can be a hyperparameter for us to tune. For example, maybe we can use the left and right 4 words, or the previous one word. This is an early successful algorithm for words embedding.

Word2vec Skip-Gram

Skip-gram is one of the unsupervised learning techniques used to find the most related words for a given word(context word).



The output y_hat would be the probability of each word would be the target word. So if we have vocab size = 10,000, the y hat for one example would be 10,000 dimension, so computational cost would be a problem for skip-gram

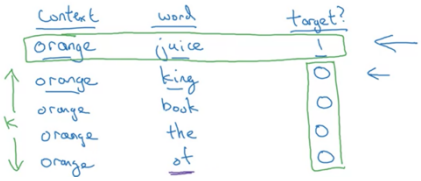


One potential solution would be hierarchical softmax.

But one more question would be, how to choose the context words to train? Obviously we don't want to choose those common words as in, a, an, on etc.

Negative Sampling

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



1. sample some context words, like "orange"
2. And find a target word for it, and label it as 1. ie, words are within the target window
3. Find k non-target words, label them as 0. Non-target words are generated by randomly selected in the example. If we accidentally selected words within the target window, it's fine.
4. Generated supervised learning, with (context, word) as pairs input, and predict target as y. the problem is, given a pair of words, predict if they are near each other.
5. K = 5-20 for smaller data, k = 2-5 for larger data

In skip-gram, the output will be 10,000 dimensional for 10,000 vocab size, but for negative sampling, for each iteration, instead of training all words in the dictionary and find its probability of being target word, we only train k+1 words, k non-target words, and 1 target words.

Selecting Negative Samples:

The non-target words are called negative samples, are usually sampled using a "unigram distribution", where more frequent words are more likely to be selected as negative samples. But the downside would

be, it will easily choose those “an, a , the” (stop words). So with some modification, the probability of choosing the word as negative samples is expressed as :

$$P(w_i) = \frac{f(w_i)^{\frac{3}{4}}}{\sum_{j=0}^{n} f(w_i)^{\frac{3}{4}}}$$

GloVe (Global vectors for word representation)

$$x_{i,j} = \#\ times\ j\ appears\ in\ context\ of\ i$$

So i is the context word and j is the target word

It is an **unsupervised** learning algorithm developed by Stanford for generating word embeddings by aggregating global word-word co-occurrence matrix from a corpus.
The model trained by the loss function :

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

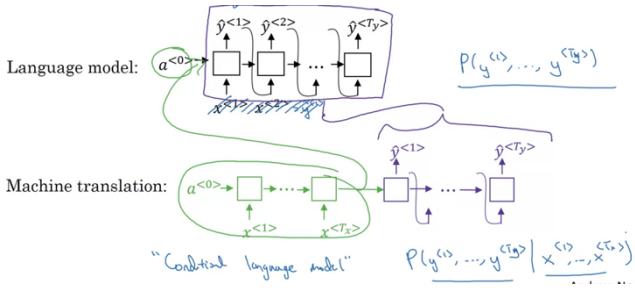
This function minimizes the difference between (how related are two words) with (how often they appear near each other)

Applications Using Word Embeddings

Sentiment Classification

A classification algorithm classifies the emotion(such as start review) of a text. The problem with sentiment classification is it usually has small number of labeled dataset, but with word embeddings, it can be solved.

Various Sequence To Sequence Architecture



Machine Translation model is combined by two parts, encoding(green) and decoding(purple). The decoding part is very similar as language model, the only difference is that instead of having a0 as input, machine translation has the encoding part for input x(such as French) then output as y(such as English). So machine translation model is called "conditional language model" as well.

Unlike language model, for machine translation, we don't only want to generate the random guess of the translation, instead, we want $\arg\max_{y} \prod_{t=1}^{T_y} P(y^{<t>} | x, y^{<1>}, \ldots, y^{<t-1>})$

but if we use greedy search, for a large dictionary, the computational cost would be high, so we need to choose carefully for our search algorithm

Beam Search

1. Set B(beam width) = number of most possible words for $x^{<1>}$, and keep them in memory
    a. Large B: better results, slower
2. For each possible word, let's set it to be $y^{<1>}$, to find the most possible word for $x^{<2>}$, calculating $P(y^{<1>}, y^{<2>}|x) = P(y^{<1>}|x)\ P(y^{<2>}|\ x, y^{<1>})$ for all 10,000 words in the dictionary, so there are B*10,000 possibilities in total
3. We find the top B possibilities to be our $x^{<2>}$. In another word, we are looking for the top B $x^{<1>}$, $x^{<2>}$ combinations.
4. And we do the same thing for all $x^{<i>}$ in this example sentence

Refinements to Beam Search

Our original objective function tends to favor shorter sentences. To solve this issue, we refined our objective function to be:

$\frac{1}{T_y^{\alpha}} \sum_{t=1}^{T_y} \log P(y^{<t>} | x, y^{<1>}, \ldots, y^{<t-1>})$ This is called length normalization. And alpha is a hyperparameter to tune

Unlike BFS, or DFS, beam search runs faster, but is not guaranteed to converge.

To minimize the problems caused by beam search, we use **Error Analysis**

Error analysis on beam search

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

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

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

Beam search chose $\hat{y}$. But y^* attains higher $P(y|x)$.

Conclusion: 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 $P(y^*|x) < P(\hat{y}|x)$.

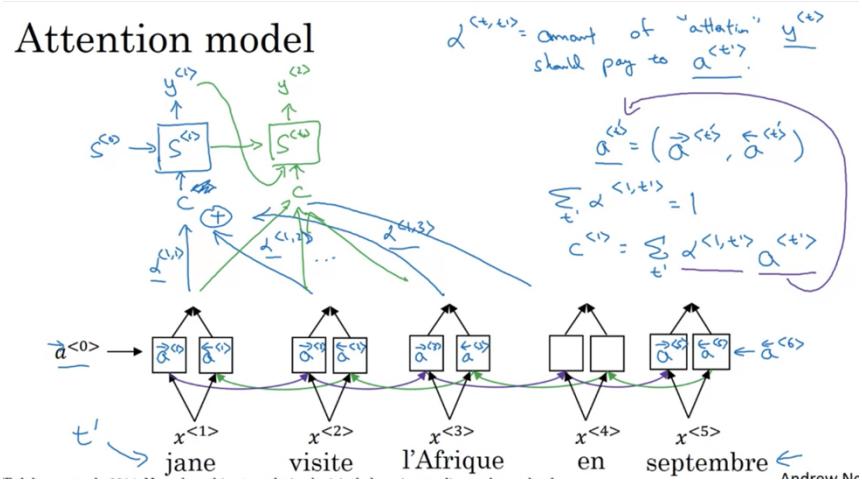
Conclusion: RNN model is at fault.

we run this error analysis for each wrong predicted example in our training, and figure out what fraction of errors are due to beam search vs. RNN model

Attention Model

For general machine translation model, it usually decreases the accuracy with longer sentences input. One solution would be using Attention Model

Attention model is the sequence model that we take into account of some nearby words in the sentence. So it's just like how our brain works. Say when we are trying to translate a long sentence, it's hard for a human being to remember the whole sentence and translate it at one time. What we will typically do is we look at part of the sentence, we translate it, and we move on. And this is basically what Attention Model is doing. Attention model is one of the breaking concept in NLP these years.



t' : any other words in the sentence other than word t

We use bidirectional RNN in the model. When we are translating the first word in ith example:

1. Input activation 0
2. We will calculate alpha, the amount of attention we should pay to, for each word in the sentence
3. Calculate $c = \sum_{t'} \alpha^{<1,t'>} a^{<t'>}$ feed to hidden state

For the next word, we will do the same thing and feed the output of first word to it as well

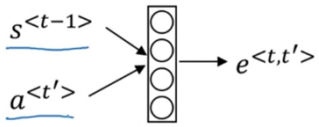
Computation Details

To compute the amount of attention $y^{<t>}$ should pay to $a^{<t'>}$

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

Then how do we compute e?

We usually fit a small NN, with 1 or two hidden layers



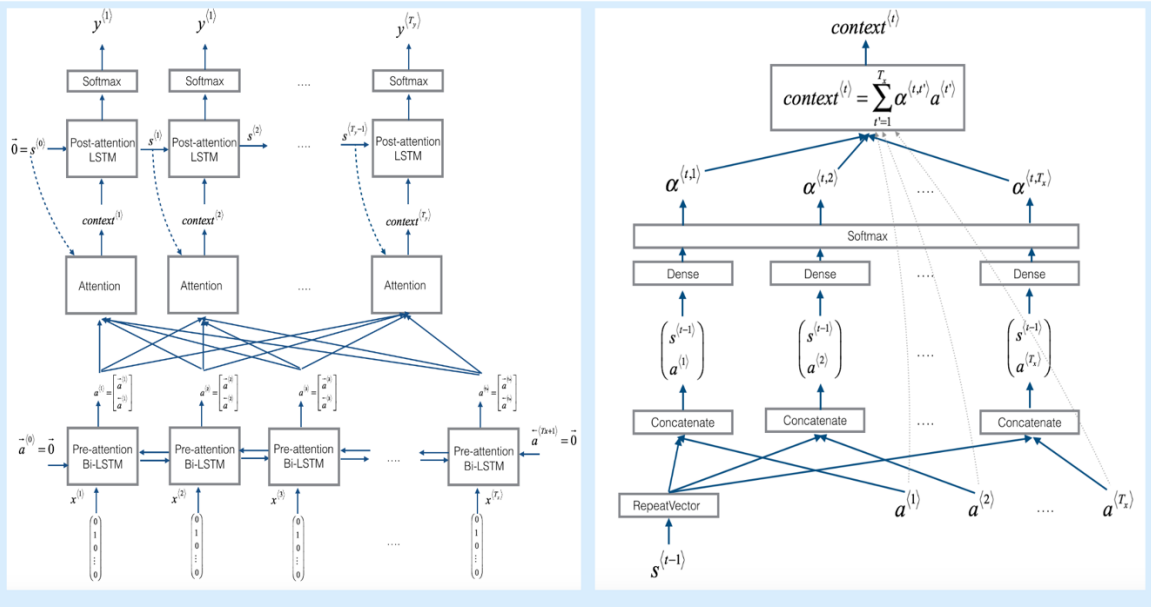
input : $S^{<t-1>}$ is the previous hidden state

$a^{<t'>}$ is the activation for the word time step t'

we trust Gradient Descent to learn the function

Cons of this algorithm is the computational cost is quadratic to other RNN

More implementation details about how each layer works



Pre-attention and Post-attention LSTMs on both sides of the attention mechanism

- There are two separate LSTMs in this model (see diagram on the left): pre-attention and post-attention LSTMs.
- **Pre-attention** Bi-LSTM is the one at the bottom of the picture is a Bi-directional LSTM and comes before the attention mechanism.
    - The attention mechanism is shown in the middle of the left-hand diagram.
    - The pre-attention Bi-LSTM goes through T_xTx time steps
- **Post-attention** LSTM: at the top of the diagram comes after the attention mechanism.
    - The post-attention LSTM goes through T_yTy time steps.
- The post-attention LSTM passes the hidden state $s\langle t\rangle$s⟨t⟩ and cell state $c\langle t\rangle$c⟨t⟩ from one time step to the next.

Here's what you should remember

- Machine translation models can be used to map from one sequence to another. They are useful not just for translating human languages (like French->English) but also for tasks like date format translation.
- An attention mechanism allows a network to focus on the most relevant parts of the input when producing a specific part of the output.
- A network using an attention mechanism can translate from inputs of length T_xTx to outputs of length T_yTy, where T_xTx and T_yTy can be different.
- You can visualize attention weights $\alpha\langle t,t'\rangle$α⟨t,t′⟩ to see what the network is paying attention to while generating each output.

Speech Recognition

Understand dataset

From Audio Recordings to Spectrograms

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 also perceives as sound.
- You can think of an audio recording as a long list of numbers measuring the little air pressure changes detected by the microphone.
- For example, if we use audio sampled at 44100 Hz (or 44100 Hertz).
    - This means the microphone gives us 44,100 numbers per second.
    - Thus, a 10 second audio clip is represented by 441,000 numbers (= $10\times44{,}10010\times44{,}100$).

Spectrogram

- It is quite difficult to figure out from this "raw" representation of audio whether the word "activate" was said.
- In order to help your sequence model more easily learn to detect trigger words, we will compute a **spectrogram** of the audio.
- The spectrogram tells us how much different frequencies are present in an audio clip at any moment in time.
- A spectrogram is computed by sliding a window over the raw audio signal, and calculating the most active frequencies in each window using a Fourier transform.

The graph above represents how active each frequency is (y axis) over a number of time-steps (x axis).

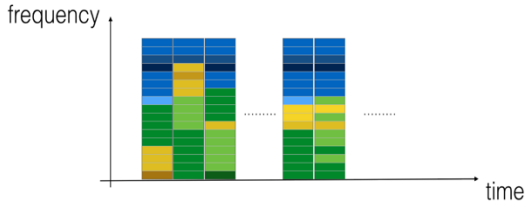


Figure 1: Spectrogram of an audio recording

- The color in the spectrogram shows the degree to which different frequencies are present (loud) in the audio at different points in time.
- Green means a certain frequency is more active or more present in the audio clip (louder).
- Blue squares denote less active frequencies.
- The dimension of the output spectrogram depends upon the hyperparameters of the spectrogram software and the length of the input.

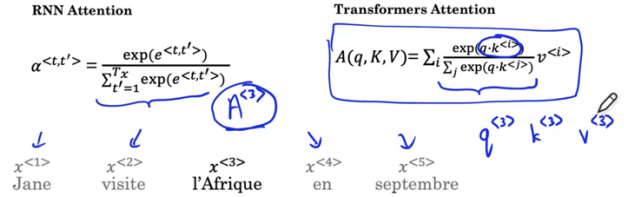
Transformers Network

A paper called “Attention Is All You Need” published in 2017 comes into the picture, it introduces an encoder-decoder architecture based on attention layers, termed as the transformer.
One main difference is that the input sequence can be passed parallelly so that GPU can be utilized effectively, and the speed of training can also be increased. And it is based on the multi-headed attention layer, vanishing gradient issue is also overcome by a large margin. The paper is based on the application of transformer on NMT(Neural Machine Translator).

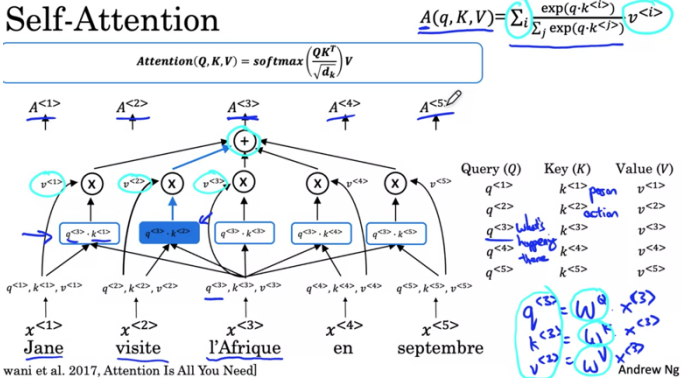
Self-Attention Intuition

$A(q,K,V)$= attention-based vector representation of a word

↳ calculate for each word $A^{<1>}, \dots, A^{<5>}$



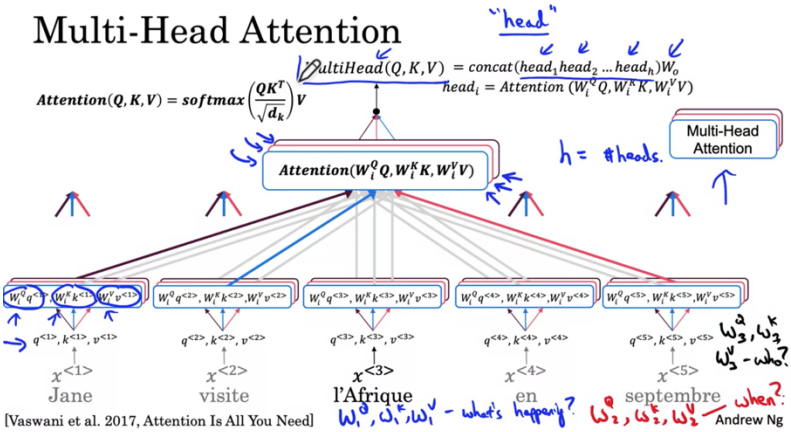
Unlike RNN Attention model, we calculate attention for each word one at a time, Transformers Attention model calculates all words’ attentions in parallel.



Here is an example of how we calculate the attention for the third word in the sequence. Query represents the question we ask about the word, such as “What’s happening there”, What we're going to do is compute the inner product between q^3 and each k, such as k^2 and this is intended to tell us how good is visite an answer to the question of what's happening in Africa and so on for the other words in the sequence. The goal of this operation is to pull up the most information that's needed to help us compute the most useful representation A^3 up here.
Q = interesting questions about the words in a sentence, K = qualities of words given a Q, V = specific representations of words given a Q

The key advantage of this representation is the word of l'Afrique isn't some fixed word embedding. Instead, it lets the self-attention mechanism realize that l'Afrique is the destination of a visite, of a visit, and thus compute a richer, more useful representation for this word.

Multi-Head Attention

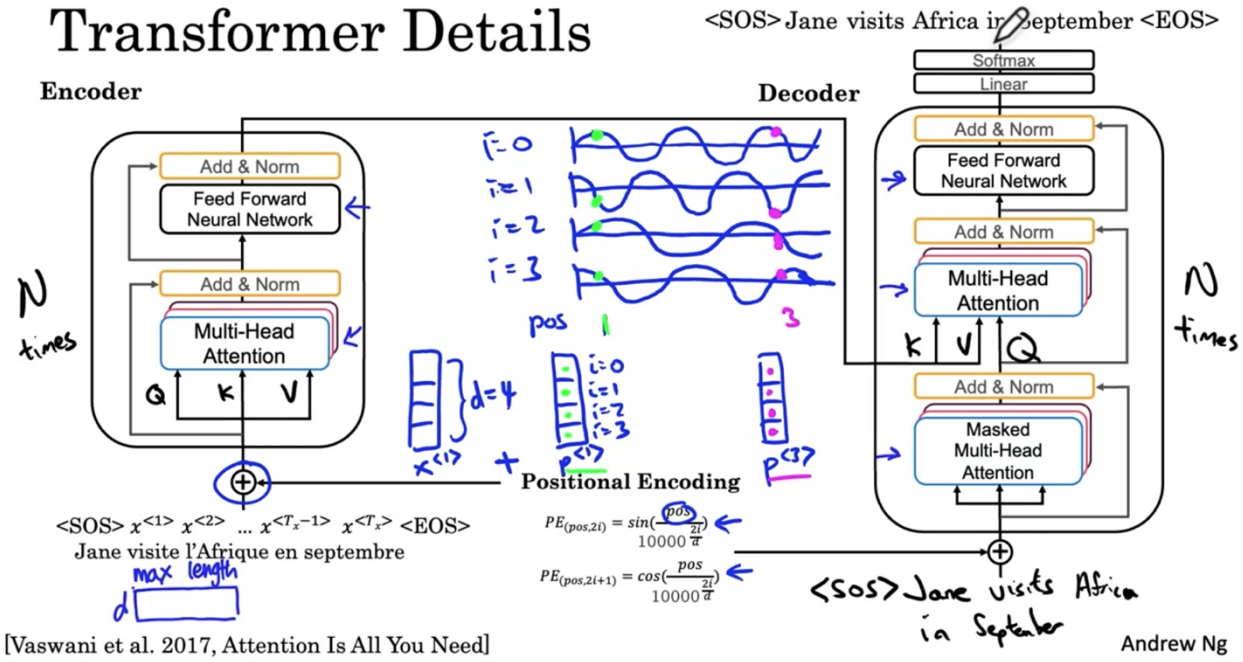


[Vaswani et al. 2017, Attention Is All You Need]

The problem Attention model faces is that for every word it weighs its value much higher on itself in the sentence, be we are inclined towards its interaction with other words of that sentence. So, we determine multiple attention vectors per word and take a weighted average, to compute the final attention vector of every word.

For multi-head attention, we asked h queries for each word and calculate the attention for each of them and stacking them together(representing by the blue, red and black squares). For each attention, we might get a different word has the highest relevance with the ith word. So we get a much richer representation of our attention vector.

Transformer Details



[Vaswani et al. 2017, Attention Is All You Need]

Andrew Ng