Write-up for MAT497

Yuling (Max) Chen Supervised by Prof Alik Sokolov

Contents

1 Intro **2**

2 RNN & LSTM **2**

2.1 RNN 2

2.2 LSTM 3

3 Attention and ARNN **4**

3.1 Neural Turing Machines 4

3.2 Attentional Interfaces 5

3.3 Adaptive Computation Time 6

3.4 Neural Programmers 7

4 Transformer Model **7**

4.1 Self-Attention 8

4.2 Layer Normalization 9

4.3 The Decoder 10

5 BERT **11**

1 Intro

This is the reading note of the course MAT497 (Research Project in Math) by Yuling Chen, suprvised by Prof Alik Sokolov. The project was inspired by the 2 papers from RiskLab, University of Toronto, "*Learning Optimal Manifold Embeddings on Financial Time Series*" (by Alik Sokolov, Jonathan Mostovoy, Brydon Parker, Luis Seco) and "*Constructing ESG Indexes Using Learned Representations of Text Data*" (by Alik Sokolov, Jack Ding, Jonathan Mostovoy, Luis Seco). Targeting the 2 models applied within the 2 papers (transformer model and BERT model), this reading course started with the basics (RNN and LSTM models), and generally stepped into the 2 targets, then, with one more step forward, went to the Xlnet model. The project of this course was based on yelp review data, and the example "*A Hands-On Guide To Text Classification With Transformer Models (XLNet, BERT, XLM, RoBERTa)*" written by Thilina Rajapakse.

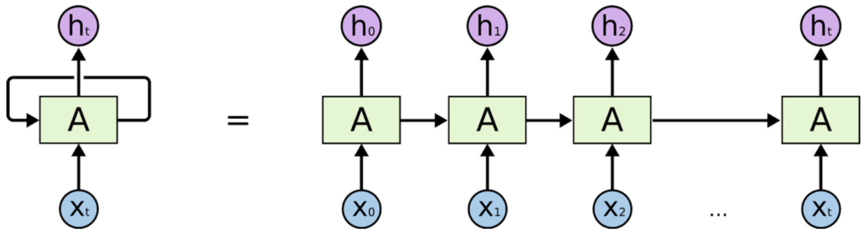
For the coding part of this course, please go to the Github page of myself for details. In the following sections of this write-up, I am going to go through what I have read and learned so far, as well as my understandings and/or comments to these deep learning models. The purpose of writting this up is to remark the path I have been through since I started researching on machine learning (especially deep learning), which might be a guidline of study resources for further readers if you are also struggling with this topic.

2 RNN & LSTM

Retrieved from "*Understanding LSTM Networks*"", by Christopher Olah.

2.1 RNN

Recurrent Neural Networks (RNNs) are no more than just a chunck of copies of neural networks. At the i-th iteration, the the chunck of neural network A takes an example x_i and returns a hidden value h_i, and this procedure keeps recurring over and over again until some condition of termination is satisfied. In practice, one can train a model to solve problems like speech recognition, language modeling, translation, image captioning, using an RNN.



An unrolled recurrent neural network.

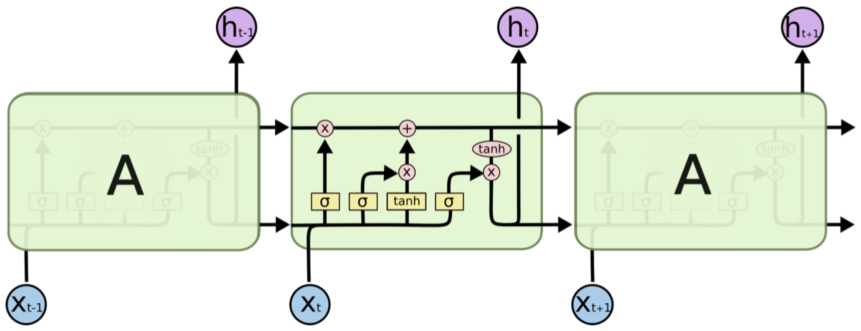
A closer glance of RNN could be like the following:

In the big picture of this project, we are aiming to train a model (classifier) to understand the context of a bunch of text sequences (e.g. yelp reviews of a merchant or tweeter posts of the ESG stock market) and classify whether it is positive or negative.

Simple RNNs are great in theory but it can be practice it can go wrong as we increase the length and complexibility of the sequences. RNNs work well on short-term dependencies but work poorly on long-term dependencies. That is, if we were to understand the context of “What a wonderful day!”, RNN can easily mark it as a positive sentiment by capturing the word “wonderful”. But if the original sequence was “I forgot to bring an umbrella in such a heavy rainy weather, and I missed the last bus to go back home. What a wonderful day!”, then RNN will mark this statement as a positive in all innocence, which is obviously wrong here. To fully understand the context of this sequence, we have to refer further back to the unfortunate experience of this person. LSTM saved us from this problem.

2.2 LSTM

Long Short Term Memory (LSTM) neworks are a special type of RNN. The LSTTM looks nothing special but this:



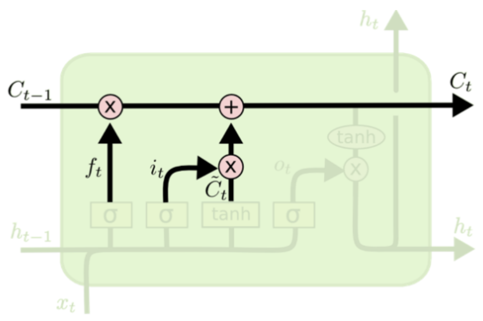
The repeating module in an LSTM contains four interacting layers.

Mathmematically, this means:

$$
\begin{aligned}
& f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f), \text{ forget gate layer} \\
& i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i), \text{ input gate layer} \\
& \tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C), \text{ a candidate value for the cell state} \\
\Longrightarrow\ & C_t = f_t \times C_{t-1} + i_t \times \tilde{C}_t, \text{ update of the cell state} \\
\Longrightarrow\ & o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o), \text{ output gate layer} \\
& h_t = o_t \times \tanh(C_t), \text{output the next hidden value}
\end{aligned}
$$

Intuitively, the LSTM model at each step t takes the hidden value from the last step (h_{t-1}) and the new data from outside (x_t) as inputs. Via the forget gate layer, the model determines how much information to be forgot from the last step (i.e. how much information to be memorized and hence passed on to the next step). Then the input gate layer decides what new information to be stored in the cell state. A candidate cell state is activated using tanh function, and in (linear) combination with the output of the input gate layer, we have the new cell state. Lastly, the output gate layer decides what information we want to output, in a filtered version. The final output of this current step is the hidden value h_t.

Different from simple RNNs, the LSTM model not only passes the hidden value on to the next step, but it also passes the cell state on to its descendants. The cell state carries the information from the further past that helps the LSTM to handle long-term dependencies, which is the core of LSTM models.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

For details please refer to the original post.

3 Attention and ARNN

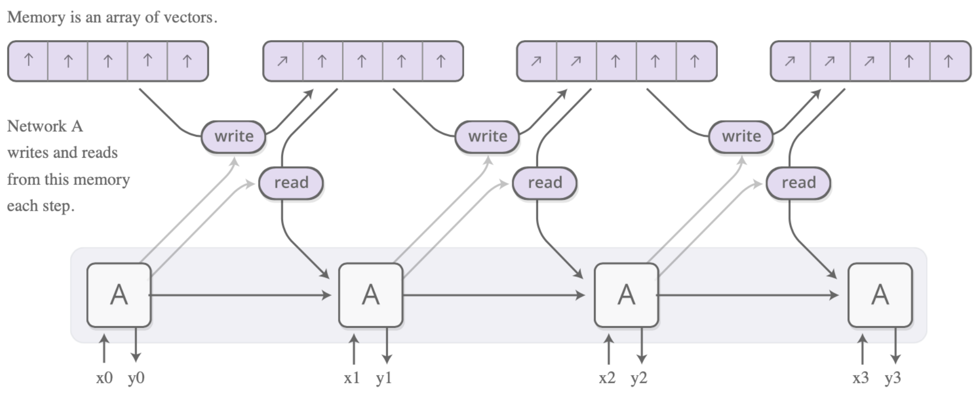
Retrieved from *Attention and Augmented Recurrent Neural Networks*, by Olah & Carter (2016).

Yes LSTM is able to capture the long-term dependencies among sequences. But to what extend shall we capture such dependencies? Imagine we are rushing for an exam tomorrow at 8:00 am, but you just came out from a bar and it is now 11:00 pm. Neither your brain nor your clock allows you to review/learn all the materials for the test, despite you know for sure that this is the best way of preparation. Given this poor situation, all you could do is to focus on the most relevant and core part of your study materials (and wish the best of luck for yourself).

The compromise of learning accuracy/efficiency and computational power/time is the nature of the Attention structure.

As described by Olah & Carter (2016), there are 4 main types of Attention structures, which can be applied to various Augmented RNNs (ARNN).

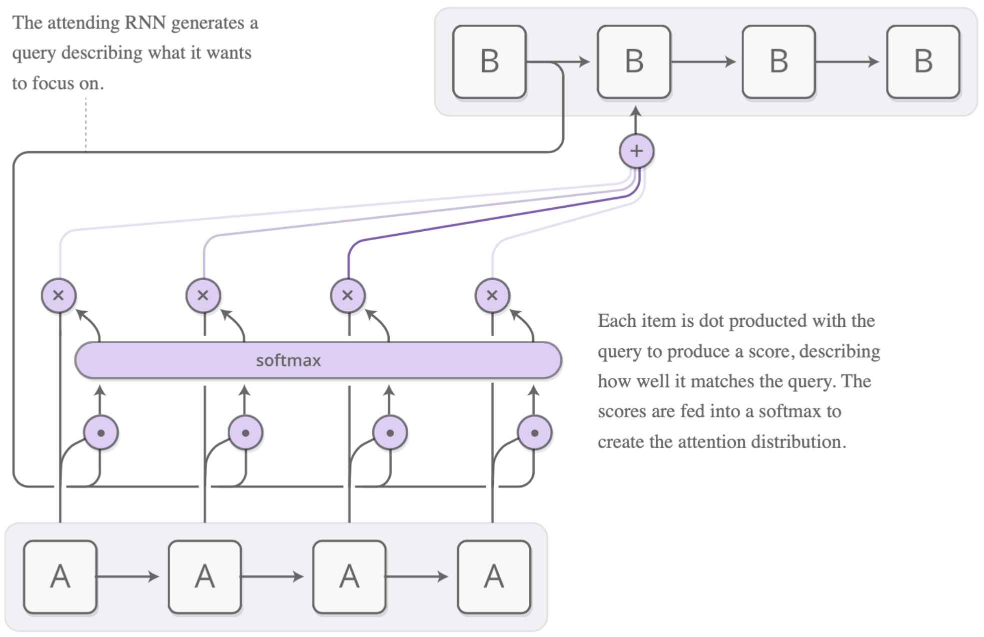
3.1 Neural Turing Machines


Memory is an array of vectors.
Network A writes and reads from this memory each step.
write
read
A
x0
y0
x1
y1
x2
y2
x3
y3

NTM combines a simple RNN with an external memory bank. For each iteration, the RNN “reads” (i.e. inputs) the memory bank and “writes” (i.e. updates) on it as a whole. Attention is performed as the extent (e.g. weights) to which each part of the memory bank is updated/modified.

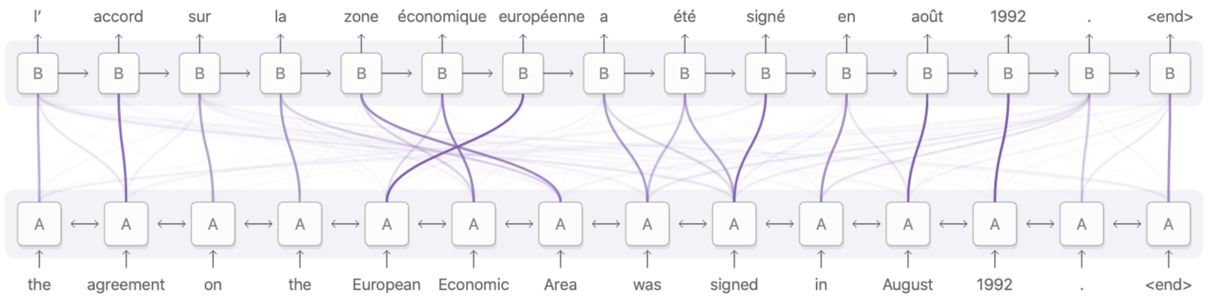
NTMs decides the position in memory to which it focuses its attention on via 2 methods: content-based attention and location-based attention. The content-based attention allows NTMs to scan over the whole memory and focus on places that match what they are looking for; while location-based attention allows relative movement in memory, enabling the NTM to loop.

3.2 Attentional Interfaces


The attending RNN generates a query describing what it wants to focus on.
B
B
B
B
+
×
×
×
×
softmax
Each item is dot producted with the query to produce a score, describing how well it matches the query. The scores are fed into a softmax to create the attention distribution.
A
A
A
A

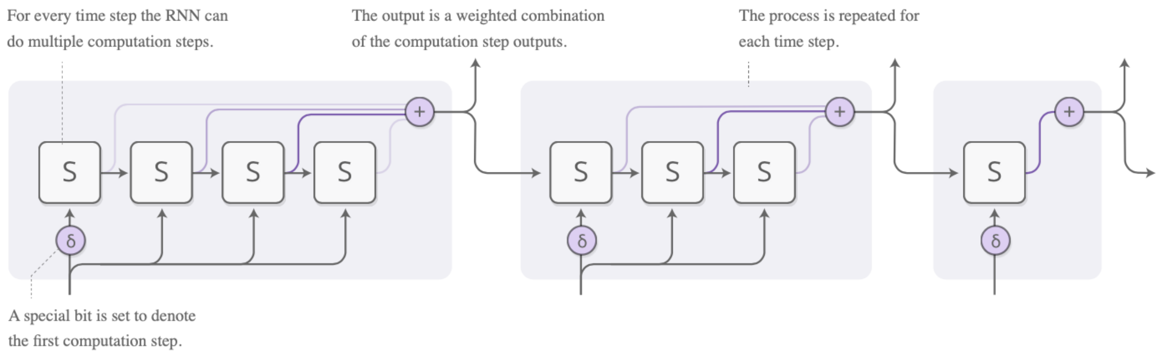
When you are reading this paper, you pay particular attention to the part you are right now reading, rather than the parts on the next page. Attention Interfaces applies this behavior by allowing RNNs to focus on the subset of information they are currently given. E.g., an RNN can attend over the output of another RNN, that is, one RNN focuses on different positions in the other RNN at every time step.
This attention structure is particular useful for sequence-to-sequence (S2S) translation. When translating a French sentence to English, we pay special attention to the word we are currently reading. So, we can apply an RNN to the French sequence, reading and understanding the content. And then feed the output of this RNN to another RNN that is applied to the English sequence.

A more intuitive illustration looks like so. The darker the line is, the higher the correlation/correspondence between the 2 words. As one can notice, the order of words in the English phrase "European Economic Area" is exactly opposite to that of the French phrase "zone économique européene", and the model correctly captures this difference and flips the direction of its attention while translating this phrase:

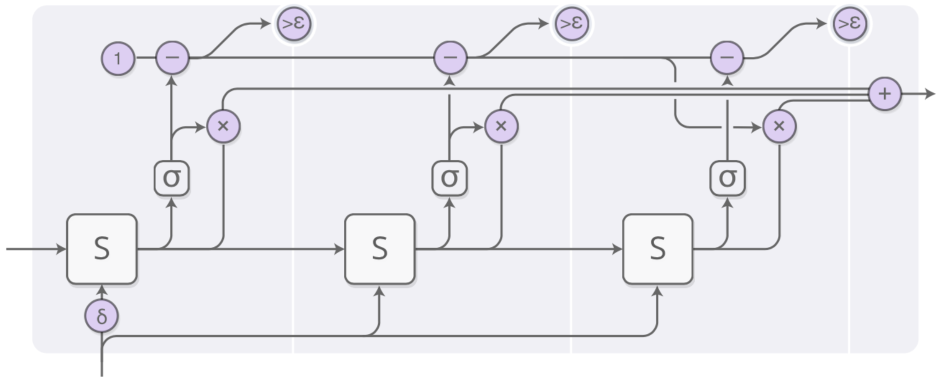


3.3 Adaptive Computation Time

If you feeling this subsection unintuitive, you might re-read this part back and forth until you fully understand. Therefore, it makes no sense for a model to be forced to complete each task using the same amount of time. Adaptive Computation Time (ACT) allows a simple RNN to spend more time/iterations on a task/sequence if it is hard.


For every time step the RNN can do multiple computation steps.
The output is a weighted combination of the computation step outputs.
The process is repeated for each time step.
S
+
δ
A special bit is set to denote the first computation step.

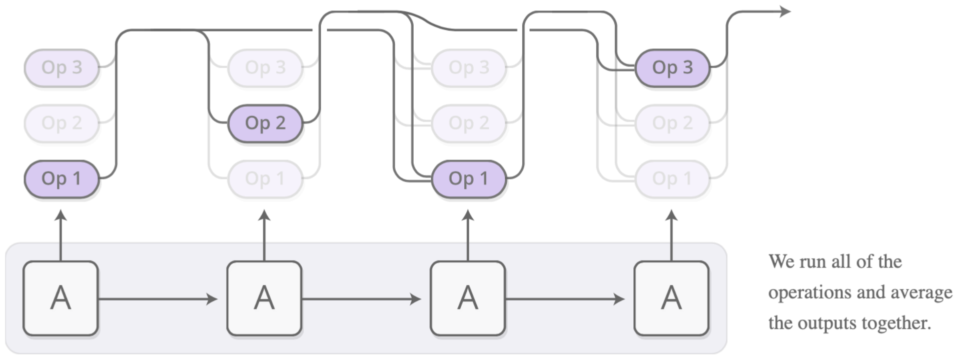
But how do we determine the number of steps/iterations to do for each task? Let's take a deeper look at a neuron.



To determine the number of steps, we have to make steps differentiable. That is, the number of steps is computed from a continuous function, rather than a discrete one. Then, each step is weighted by a "halting neuron" (σ), which is a sigmoid neuron that looks at the RNN state and gives a halting weight. This can be viewed as the probability that we should stop at this step.

The total budget of the halting weight is 1. For each step (S) of training, we subtract the halting weight of this step from the remaining weight budget of the previous steps. Once the remaining halting budget is less than a threshold (ϵ), we stop training for this taks and move forward to the next.

3.4 Neural Programmers


Op 3
Op 2
Op 1
A
We run all of the operations and average the outputs together.

Given a set of operations, Neural Programmers allow the RNN to find the best order/combination of operations to solve a problem. This is achieved by attending to different operations at each iteration.

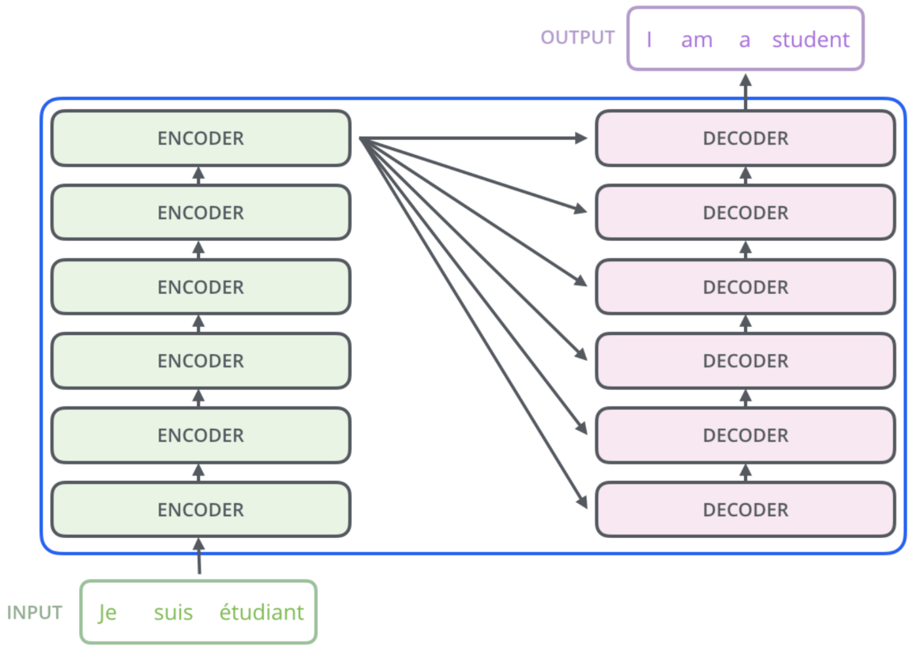
For further details, please refer to the original post.

4 Transformer Model

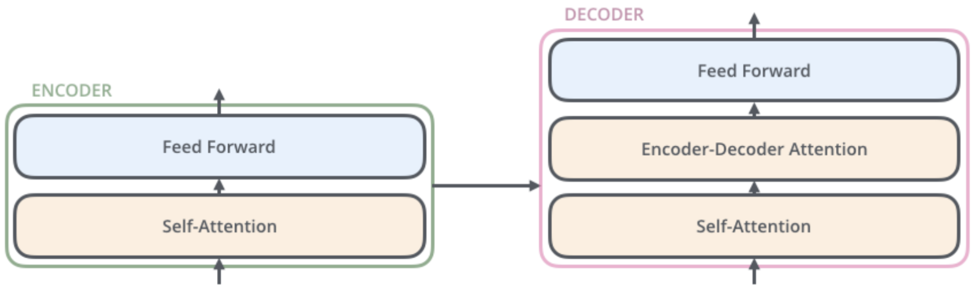
Retrived from *The Illustrated Transformer* and *Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention)*, by Jay Alammar.

The Transformer Model was proposed in the *Attention is All You Need*, by Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin (2017). And a great PyTorch illustration can be found here. The Transformer model is a great extension of the Attention structure, that has seized the crown of LSTM in the past 10 years.

The big picture of Transformer is nothing but a chunk of interacting encoders and decoders:



Each encoder has a Self-Attention layer and a Feed Forward layer, and each decoder also has the 2 layers and plus a Encoder-Decoder Attention layer in between.



4.1 Self-Attention

Self-Attention is the method the Transformer uses to hook up the "understanding" of other relevant words with the one we are currently processing. The algorithm is performed as follows:

(1) Create 3 vectors/matrices: Query, Key, and Value, using their corresponding weights (W^Q, W^K, W^V) we have trained so far. (For the very first step, initialize the weights)

$$Q = W^Q \cdot X$$
$$K = W^K \cdot X$$
$$V = W^V \cdot X$$

where X is the embedding matrix and each row of X is an embedding of each word. (Words in a sequence can be embedded into a bunch a tokens via a tokenizer. E.g. The word "reading" can be embedded as a combination of 2 tokens: "read" and "ing". This is particularly remarkable as it saves the model from exploding its memory/vocabulary bank. The model can "understand" the word "reading" by simply understand the 2 tokens, instead of having to create a new memory.)

(2) Compute a softmax standadized Score.

$$Score = Softmax(\frac{Q \cdot K}{\sqrt{dim(K)}})$$

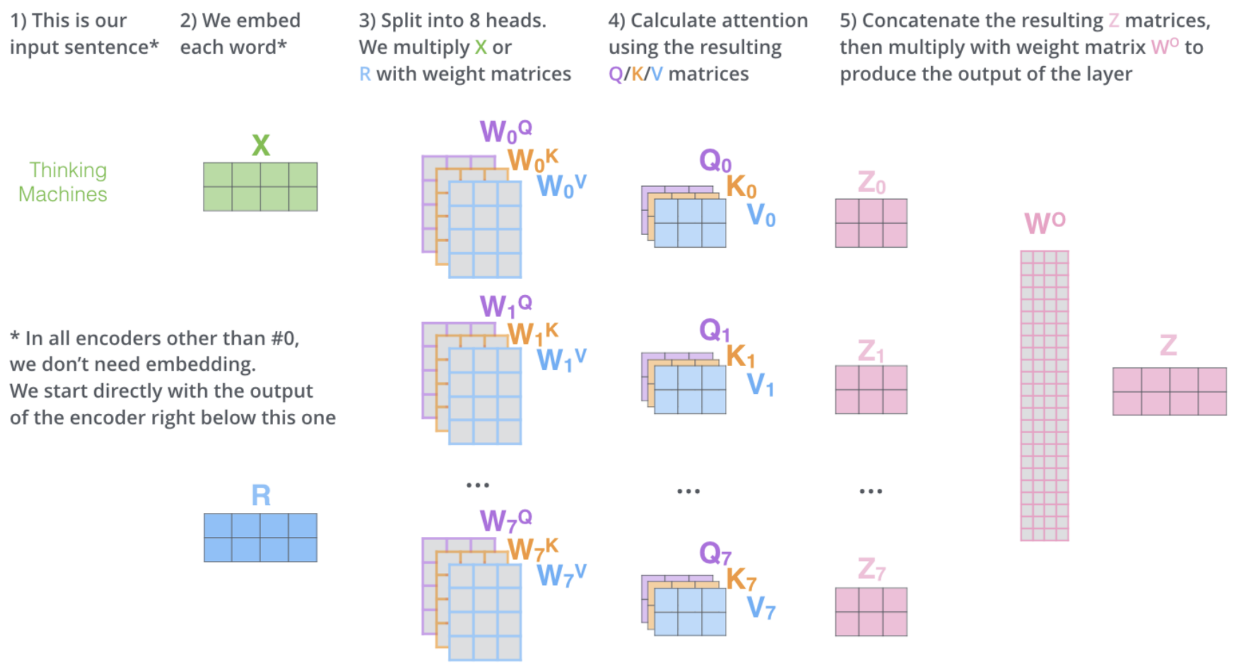
We standardize the score $(Q \cdot K)$ by dividing the square root of the dimension of the Key vector/matrix, in order to have more stable gradients. The softmax score determines how much each word will be expressed at this position. Obviously the current word at this position has the highest softmax score, but some other words at different positions may be of high relevance to the word we are currently processing. Therefore, it is very meaningful to put some attention to such words.

(3) Sum the softmax scores weighted by the value (V).

$$Z = Sum(Score \times V), \text{ output of the Self-Attention layer}$$

4.1.1 Multi-head Attention

This is a duplication of a bunch of Self-Attention layers. Each head is a Self-Attention layer, which output a Z_i. Then the final output of the Multi-head Attention layer is simply the weighted average of all the $Z_i's$.

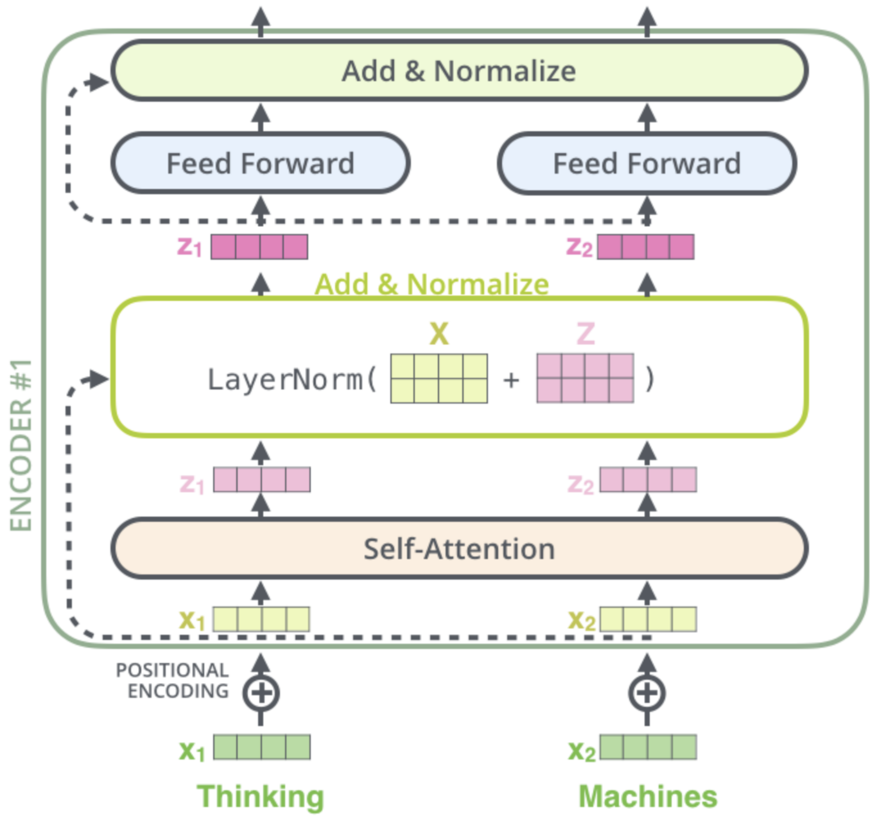


1) This is our input sentence*
2) We embed each word*
3) Split into 8 heads. We multiply X or R with weight matrices
4) Calculate attention using the resulting Q/K/V matrices
5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer
Thinking Machines
X
R
* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one
W0Q W0K W0V
W1Q W1K W1V
W7Q W7K W7V
Q0 K0 V0
Q1 K1 V1
Q7 K7 V7
Z0
Z1
Z7
WO
Z

4.2 Layer Normalization

Retrieved from *Layer Normalization*, by Jimmy Lei Ba, Jamie Ryan Kiros, Geoffrey E. Hinton (2016).

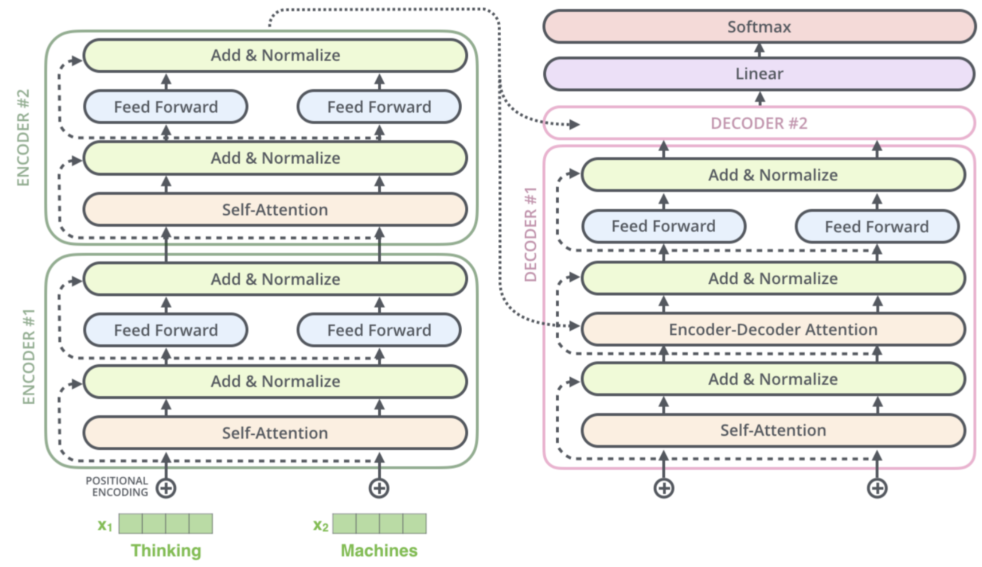
This is an add-on layer between the Self-Attention layer and the Feed-Forward layer inside the decoder. The basic idea is to reduce training time and increase computational efficiency, by "computing the mean and variance used for normalization from all of the summed inputs to the neurons in a layer on a single training case".



The positional encoding determines the position of each word, or the distance between different words in the sequence. Adding these vectors/matrices to the word embeddings helps the Transformer to understand the distances between the embedded words once we project them into the Q/K/V space.

4.3 The Decoder

Once we encoded the input, the output of the top (i.e. the last) encoder is transformed into a set of attention vectors/matrices K and V, which are fed into the Encoder-Decoder Attention layer of each decoder. The Encoder-Decoder Attention layer is pretty much the same as a Multi-head Attention layer, despite it takes the K and V from the top encoder and creates the Q from the layer below it.



The Self-Attention layer in the decoder here is only allowed to attend to earlier positions in the output sequence. The future positions are masked before the softmax step in the Self-Attention calculation. BERT model can be used to predict the masked positions.

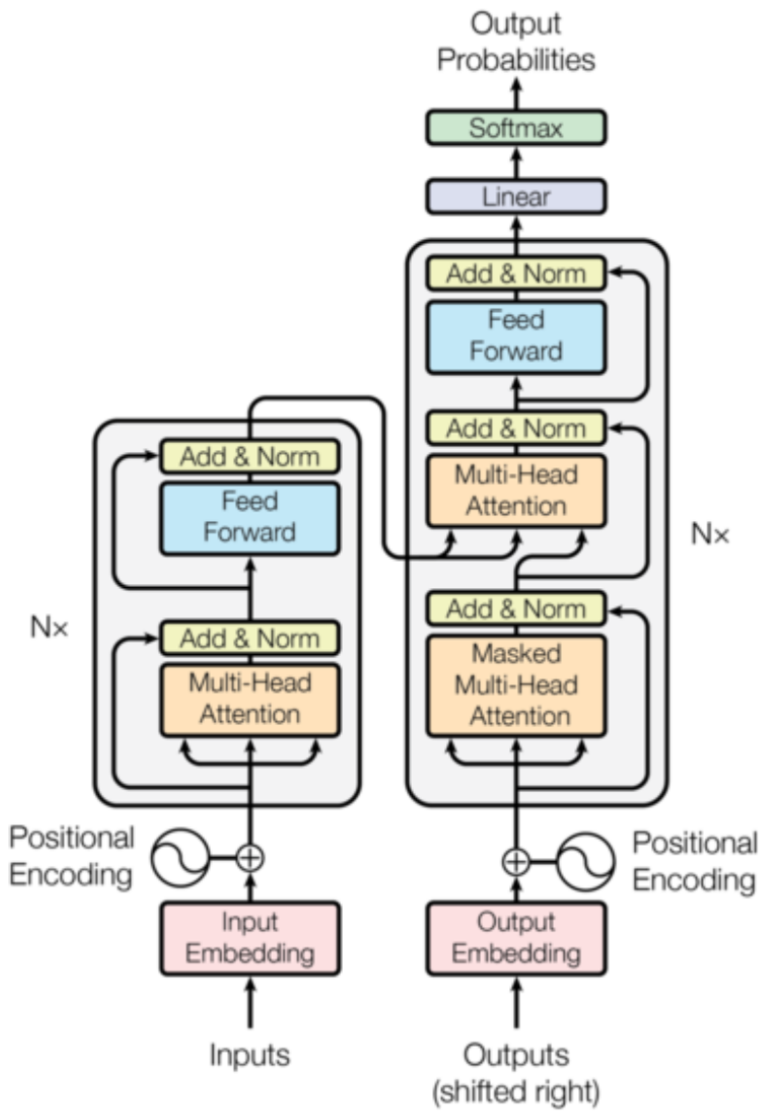
Finally, the decoders output vectors of floats, which are taken by the Linear and Softmax layer and transformed back to the word sequences. (In our case here, they are translated to French words.) The Linear layer is just a fully connected neural network, which generates the output of the decoders to a long vector called "logit vector". If our vocabulary bank contains 10000 French words, then the logit vector has length 10000. Then the Softmax layer transformed the logits into log probaibilities, and the word with the highest probability is chosen as the output for this translation.

5 BERT

Retrieved from BERT, *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding* by Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova (2018), the lecture note of Prof Jacob Devlin and the MMF lecture notes of Prof Alik Sokolov.

The Bidirectional Encoder Representations from Transformers (BERT) model was first proposed by Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova in 2018. "It is a bidirectional transformer pre-trained using a combination of masked language modeling objective and next sentence prediction on a large corpus comprising the Toronto Book Corpus and Wikipedia".

The model architecture of BERT is very similar to (if no exactly the same as) the Transformer model we have discussed above. The Feed Forward layer is a set of multilayer perceptrons that can compute the non-linear hierarchical features. For detailed explanation of Feed Forward layers, please read *Deep Learning: Feedforward Neural Networks Explained*, by NiranjanKumar (2019).



BERT is a innovation compared to the past unidirectional models, e.g. LSTM (*Semi-Supervised Sequence Learning*, Google, 2015), ELMo (*Deep contextualized word representations, AI2 & Univeristy of Washington, 2017), Deep Transformer Language Model (*Improving Language Understanding by Generative Pre-Training*, OpenAI, 2018), etc. Unidirectional models are constrained by the directionality as they need it to generate a well-formed probability distribution, which is not required by BERT. In fact, language understanding is bidirectional, rather than unidirectional (either left context or right context).