Topic 10: Sequence-to-sequence

One of the more exciting uses of neural networks is in sequence-to-sequence generation, abbreviated to seq2seq. This is when the neural network takes in a sequence as an input and returns a different sequence as an output, such as in machine translation. The input sentence is called the source sentence and the output sentence is called the target sentence. The source sentence is usually first encoded into some kind of vector representation which then conditions a language model to generate the target sentence.

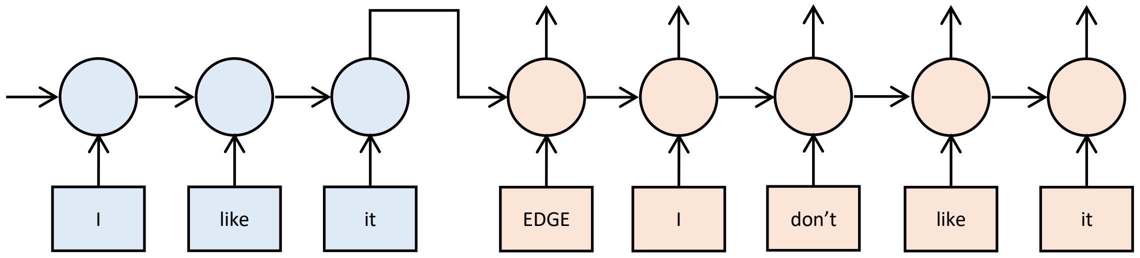
In the code we will be looking at translating the sentiment of a sentence. This is a situation where generating the most probable sentence is important (rather than sampling one randomly) as there is only one correct output sentence for each input sentence so we will be using greedy search to generate the sentences.

1. Basic seq2seq

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/10_-_Sequence-to-sequence/01_-_Basic_seq2seq.py

The traditional way to do a seq2seq neural network is to use an RNN to encode a source sentence and then use something like init-inject in order to condition a language model to generate a target sentence. This is what was done in this paper in 2014. Interestingly, the author had found that reversing the source sentence prior to encoding it results in better performance. Why do you think this is and what can we do to make it even better?

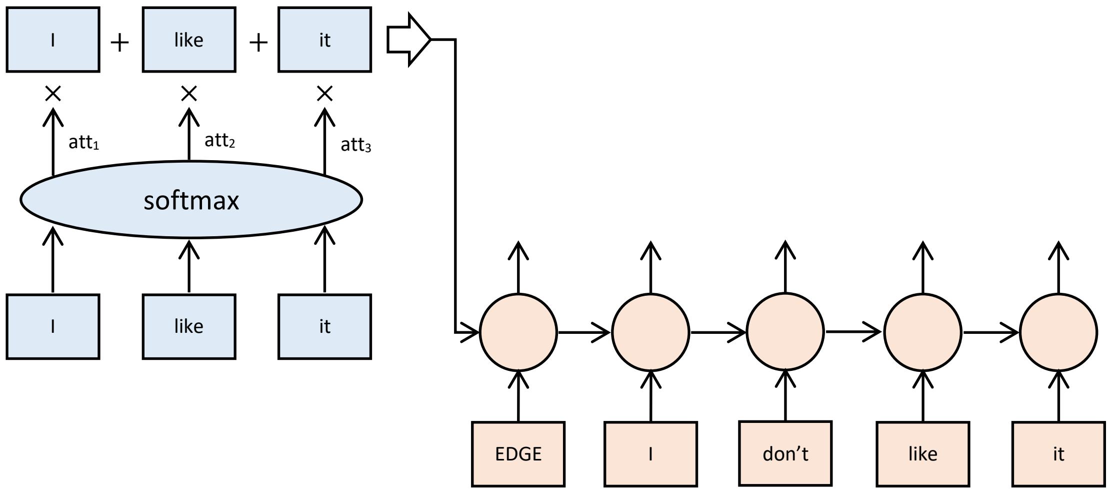
Below is an illustration of this system with the source handling part of the model in blue and the target handling part (the language model) in orange.



2. Self-attention

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/10_-_Sequence-to-sequence/02_-_Self-attention.py

The traditional way of doing seq2seq did not stick for long as in 2015 there was a significant improvement given to it, which was attention. The simplest form of attention is self-attention. When we read a sentence, we do not give equal importance to each word. Some words require more attention than others do. We can simulate attention by using a sub neural network to give a score for each word (based on the word's vector) and then take the softmax of the scores. This will result in an attention vector which sums to 1. In order to give the specified amount of attention to each word, we take a weighted average of the word vectors using the attention vector as weights.



In practice the word vectors used in attention would not be the word embeddings but instead be the outputs of a bi-directional RNN. This would encode a context for each word which would allow the neural network to determine the importance of the words in context.

Given that the source sentences are of different sizes, it is important that the attention vectors be somehow padded to accommodate any source sentence length. The problem is that the softmax function that produces the attention vector takes all the numbers in the logits vector into account when producing its output. This means that we need to somehow pad the logits with pad values that don't effect the softmax. Values that softmax ignores are very large negative numbers such as negative one million. By replacing logit values obtained from pad words with large negative numbers, softmax will give a probability of zero to those pad values. The replacement is done using a binary mask.

The nice thing about attention is that you can then check what the attention vector was like when encoding the sentence, thereby giving you a measure of how important each word was considered by the neural network and letting you inspect what is happening internally.

Note that there is no reason for the top and bottom blue "I like it" to consist of the same vectors. The vectors that determine the attention need not be the same as what is being attended to. It could be that a neural layer is used to transform the bottom word vectors into the top word vectors. In this case, the bottom vectors are referred to as the key vectors and the top vectors as the value vectors.

Finally, when generating an attention vector from a word, it is important that no less than two neural layers are used. If just one neural layer is used then the following sections will not work as the softmax will end up producing the same attention values over and over.

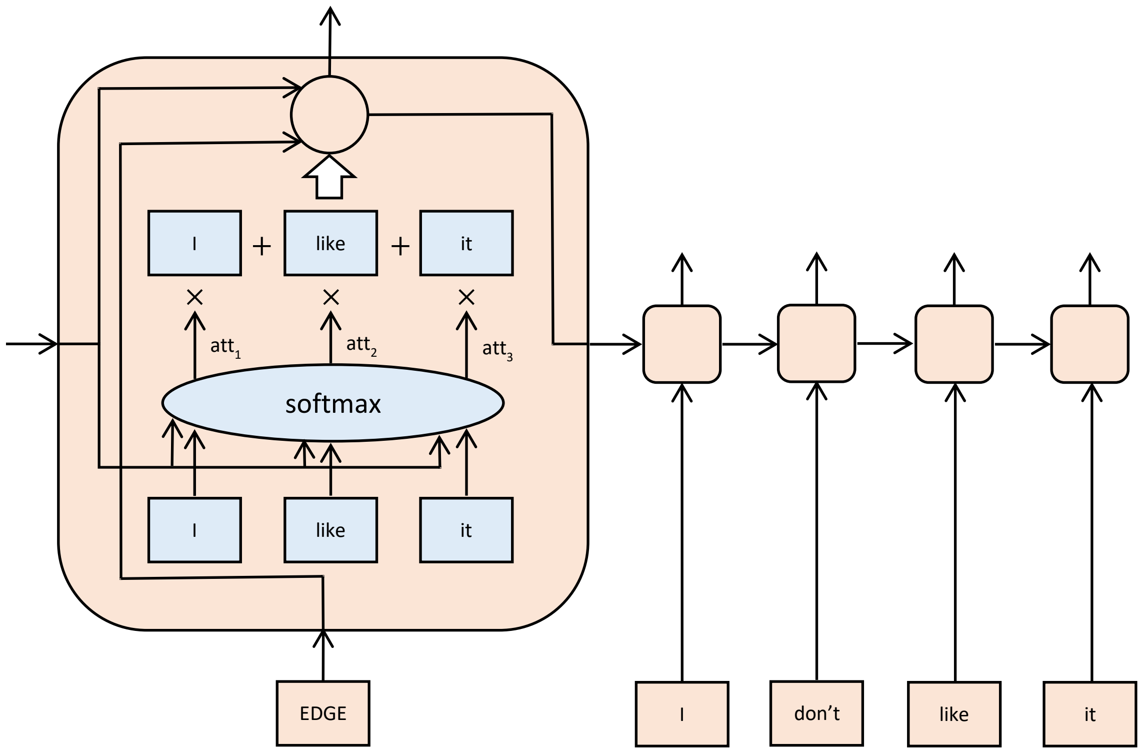
3. Dynamic attention with par-inject

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/10_-_Sequence-to-sequence/03_-_Dynamic_attention_with_par-inject.py

Self-attention is not what transformed seq2seq neural networks, it was dynamic attention, which is when the attention changes for each word being generated. This is comparable to rereading the source sentence for every word being generated, focusing on certain words more than others each time. This sort of strategy of creating different encodings for each state in the language model requires a language model conditioning method that accepts several conditioning vectors, that is, par-inject or merge. Here we will focus on par-inject which is the most commonly used method. This method was first described in this paper.

The only conceptual difference between self-attention and dynamic attention is that the attention sub-network does not produce a score based on the word vector only, but also on the current state of the language model. This means that we need to supply two inputs to the sub-network, which is just a matter of concatenating the source-word and language model state vectors together.

In practice, however this is not a trivial process to implement. First of all, since the resulting attended vector is changing the language model RNN's state, the process of attention needs to be part of the RNN's process. The next state of the RNN cannot be determined before the current state has been calculated based on the attended vector, which itself requires the previous state to compute. This means that we need to use a custom RNN cell to implement it. Below is a diagram showing what one cell would look like.



The orange circle inside the orange cell at the top is some kind of RNN cell which turns the current input and previous state into the next state as usual. You can use a GRU cell here for example (but you must use it as a single step function call within the custom RNN cell, not inside a dynamic_rnn function).

You can now investigate the attention vector as each word is generated in order to see which words in the source sentence are being 'looked at' as each word in the target sentence is being generated. This leads to a sort of alignment which lets you now which words were translated into which other words for example. The fact that the attention changes with each word also has the advantage of using the source RNN state's memory more efficiently since it doesn't need to store everything about the whole source sentence. Instead it just needs to store what is necessary for the next word to generate. In fact these sort of seq2seq models work better on very long source sentences than the traditional seq2seq model.

We have now introduced a query vector to the attention mechanism, which is the vector that contains information about what should be looked at. The query is the language model's RNN state, as that information is what determines what needs to be generated next. The key and value vectors are still the source sentence's word vectors. Also, as already stated for the self-attention model, the word vectors would not typically be the embedding vectors but instead the output vectors of a bi-directional RNN going over the source sentence.

4. Dynamic attention with merge

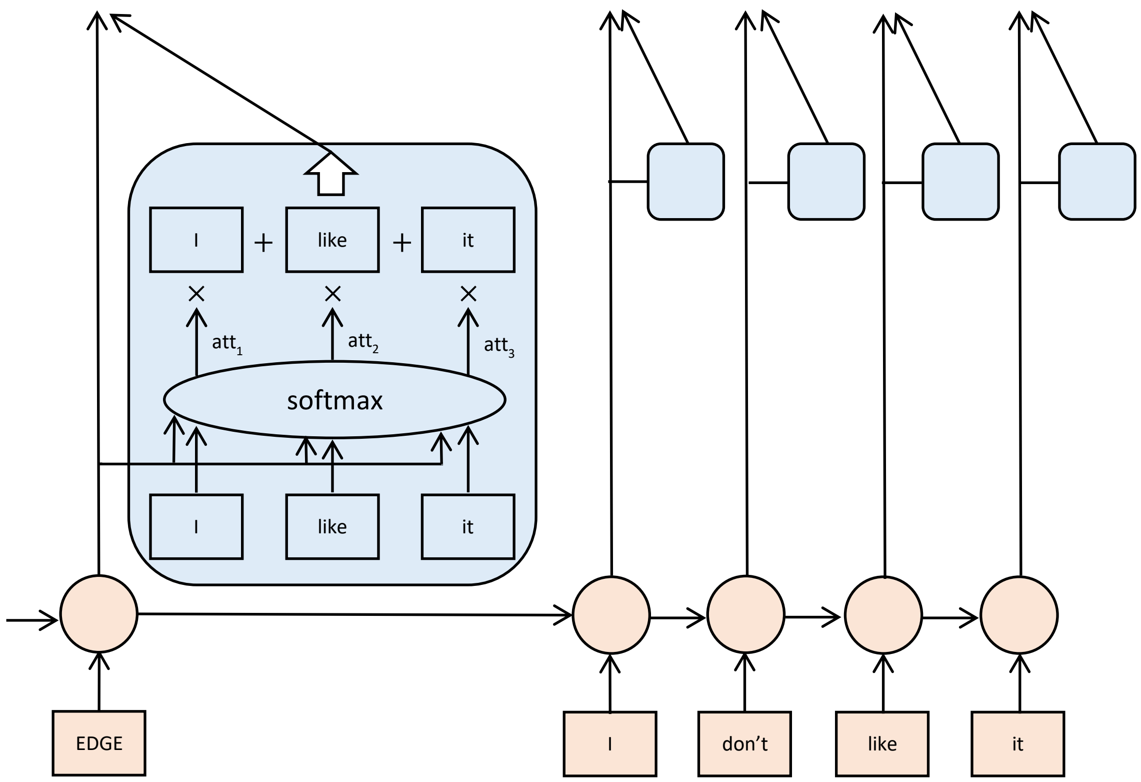
https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/10_-_Sequence-to-sequence/04_-_Dynamic_attention_with_merge.py

Of course par-inject is not the only way to condition a language model multiple times. There is also the merge architecture. The merge architecture inserts the conditioning vector after the RNN which means that the source sentence would not influence the RNN in any way, which further means that the attention mechanism does not need to be included inside a custom RNN cell.

Attention on merge is a little more code intensive than attention on par-inject as in par-inject there is only one language model state to handle at a time. With merge you need to handle all language model states together with requires you to create the following table of vectors:

	I	like	it
EDGE	$\text{att}_{\text{EDGE I}}$	$\text{att}_{\text{EDGE like}}$	$\text{att}_{\text{EDGE it}}$
I	$\text{att}_{\text{I I}}$	$\text{att}_{\text{I like}}$	$\text{att}_{\text{I it}}$
don't	$\text{att}_{\text{don't I}}$	$\text{att}_{\text{don't like}}$	$\text{att}_{\text{don't it}}$
like	$\text{att}_{\text{like I}}$	$\text{att}_{\text{like like}}$	$\text{att}_{\text{like it}}$
it	$\text{att}_{\text{it I}}$	$\text{att}_{\text{it like}}$	$\text{att}_{\text{it it}}$

Note that the words in orange are the language model RNN states resulting from those words and not some kind of word vectors like the words in blue. For every possible source word / language model state pair, there needs to be an attention vector. This is accomplished by a combination of reshaping and tiling the language model output vectors and the source sentence word vectors. After this is done, each vector is used to produce a weight average as usual which is then concatenated with the language model's output vectors prior to being passed on to the softmax.


I
like
it
att$_1$
att$_2$
att$_3$
softmax
I
like
it
EDGE
I
don't
like
it