Intuitive Understanding of Attention Mechanism in Deep Learning

A TensorFlow Implementation of Neural Machine Translation with Attention


Harshall Lamba
Mar 20, 2019 · 11 min read ★

Caution

This is a slightly advanced tutorial and requires basic understanding of sequence to sequence models using RNNs. Please refer my earlier *blog here* wherein I have explained in detail the concept of Seq2Seq models.

Table of Contents

1. Introduction

Attention is one of the most influential ideas in the Deep Learning community. Even though this mechanism is now used in various problems like image captioning and others,it was initially designed in the context of Neural Machine Translation using Seq2Seq Models. In this blog post I will consider the same problem as the running example to illustrate the concept. We would be using attention to design a system which translates a given English sentence to Marathi, the exact same example I considered in my earlier blog.

So what's wrong with seq2seq models?

The seq2seq models is normally composed of an encoder-decoder architecture, where the encoder processes the input sequence and encodes/compresses/summarizes the information into a context vector (also called as the "thought vector") of a fixed length. This representation is expected to be a good summary of the entire input sequence. The decoder is then initialized with this context vector, using which it starts generating the transformed output.

A critical and apparent disadvantage of this fixed-length context vector design is the incapability of the system to remember longer sequences. Often is has forgotten the earlier parts of the sequence once it has processed the entire the sequence. The attention mechanism was born to resolve this problem.

Let's break this down into finer details. Since I have already explained most of the basic concepts required to understand Attention in my previous *blog*, here I will directly jump into the meat of the issue without any further adieu.

2. The central idea behind Attention

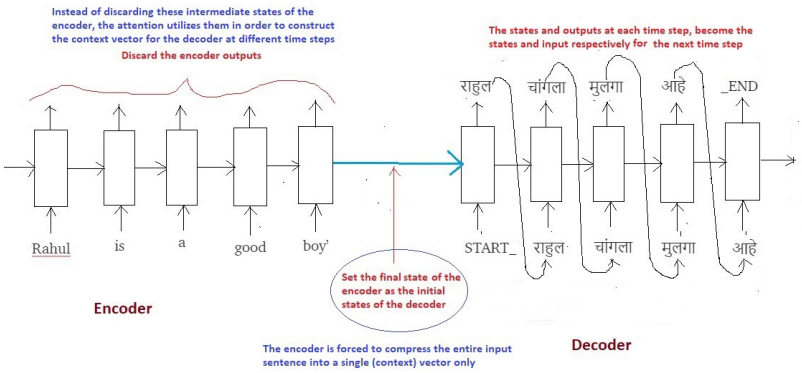
For the illustrative purposes, I will borrow the same example that I used to explain Seq2Seq models in my previous *blog*.

Input (English) Sentence: “Rahul is a good boy”

Target (Marathi) Sentence: “राहुल चांगला मुलगा आहे”

The only change will be that instead of an LSTM layer that I used in my previous explanation, here I will use a GRU layer. The reason being that LSTM has two internal states (hidden state and cell state) and GRU has only one internal state (hidden state). This will help simplify the the concept and explanation.

Recall the below diagram in which I summarized the entire process procedure of Seq2Seq modelling.

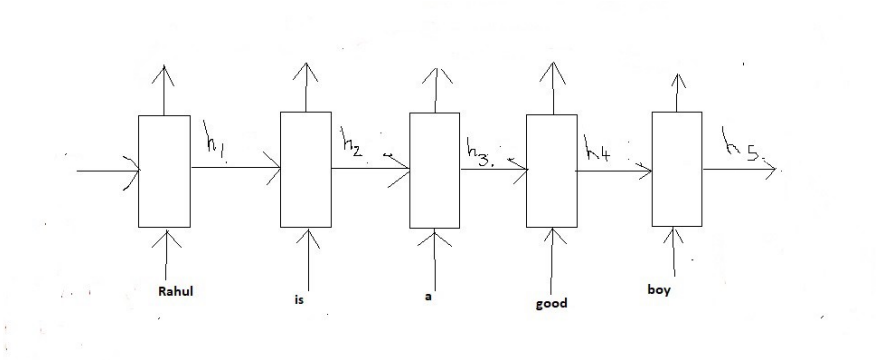


In the traditional Seq2Seq model, we discard all the intermediate states of the encoder and use only its final states (vector) to initialize the decoder. This technique works good for smaller sequences, however as the length of the sequence increases, a single vector becomes a bottleneck and it gets very difficult to summarize long sequences into a single vector. This observation was made empirically as it was noted that the performance of the system decreases drastically as the size of the sequence increases.

The central idea behind Attention is not to throw away those intermediate encoder states but to utilize all the states in order to construct the context vectors required by the decoder to generate the output sequence.

3. Why the name Attention?

Let’s name each of the intermediate states of the encoder as below:



Encoder GRU

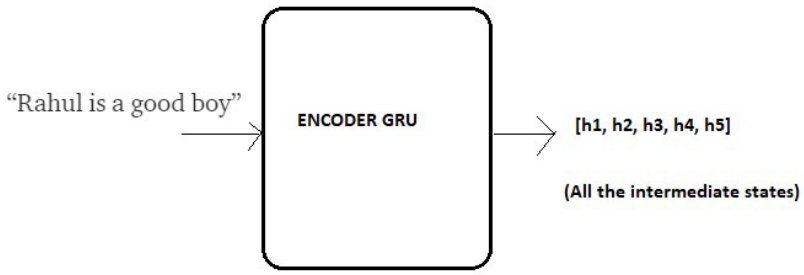
Notice that since we are using a GRU instead of an LSTM, we only have a single state at each time step and not two states, which thus helps to simplify the illustration.

Also note that attention is useful specially in case of longer sequences but for the sake of simplicity we will consider the same above example for illustration.

Recall that these states (h1 to h5) are nothing but vectors of fixed length. To develop some intuition think of these states as vectors which store local information within the sequence. For example;

h1 stores the information present in the start of the sequence (words like 'Rahul' and 'is') while h5 stores the information present in the later part of the sequence (words like 'good' and 'boy').

Lets represent our Encoder GRU with the below simplified diagram:



Compact Representation of Encoder GRU

Now the idea is to utilize all of these local information collectively in order to decide the next sequence while decoding the target sentence.

Imagine you are translating "Rahul is a good boy" to "राहुल चांगला मुलगा आहे". Ask yourself, how do you do it in your mind?

When you predict "राहुल", its obvious that this name is the result of the word "Rahul" present in the input English sentence regardless of the rest of the sentence. We say that while predicting "राहुल", ***we pay more attention*** to the word "Rahul" in the input sentence.

Similarly while predicting the word "चांगला", we pay more attention to the word "good" in the input sentence.

Similarly while predicting the word "मुलगा", we pay more attention to the word "boy" in the input sentence. And so on..

Hence the name ***"ATTENTION".***

As human beings we are quickly able to understand these mappings between different parts of the input sequence and corresponding parts of the output sequence. However its not that straight forward for artificial neural network to automatically detect these mappings.

Thus the Attention mechanism is developed to ***"learn"*** these mappings through Gradient Descent and Back-propagation.

4. How does Attention work?

Let's get technical and dive into the nitty gritty of Attention mechanism.

Decoding at time step 1

Continuing the above example, let's say we now want our decoder to start predicting the first word of the target sequence i.e. "राहुल"

At time step 1, we can break the entire process into **five steps** as below:

Decoding at time step 1

Before we start decoding, we first need to encode the input sequence into a set of internal states (in our case h1, h2, h3, h4 and h5).

Now the hypothesis is that, the next word in the output sequence is dependent on the current state of the decoder (decoder is also a GRU) as well as on the hidden states of the encoder. Thus at each time step, we consider these two things and follow the below steps:

Step 1 — Compute a score each encoder state

Since we are predicting the first word itself, the decoder does not have any current internal state. For this reason, we will consider the last state of the encoder (i.e. h5) as the previous decoder state.

Now using these two components (all the encoder states and the current state of the decoder), we will train a simple feed forward neural network.

Why?

Recall we are trying to predict the first word in the target sequence i.e. "राहुल". As per the idea behind attention, we do not need all the encoder states to predict this word, but we need those encoder states which store information about the word "Rahul" in the input sequence.

As discussed previously these intermediate encoder states store the local information of the input sequence. So it is highly likely that the information of the word "Rahul" will be present in the states, let's say, h1 and h2.

Thus we want our decoder to pay more attention to the states h1 and h2 while paying less attention to the remaining states of the encoder.

For this reason we train a feed forward neural network which will **learn** to identify relevant encoder states by generating a high score for the states for which attention is to be paid while low score for the states which are to be ignored.

Let s1, s2, s3, s4 and s5 be the scores generated for the states h1, h2, h3, h4 and h5 correspondingly. Since we assumed that we need to pay more attention to the states h1 and h2 and ignore h3, h4 and h5 in order to predict "राहुल", we expect the above

neural to generate scores such that s1 and s2 are high while s3, s4 and s5 are relatively low.

Step 2— Compute the attention weights

Once these scores are generated, we apply a softmax on these scores to produce the attention weights e1, e2, e3 ,e4 and e5 as shown above. The advantage of applying softmax is as below:

a) All the weights lie between 0 and 1, i.e., $0 \leq e1, e2, e3, e4, e5 \leq 1$

b) All the weights sum to 1, i.e., $e1+e2+3+e4+e5 = 1$

Thus we get a nice probabilistic interpretation of the attention weights.

In our case we would expect values like below: (just for intuition)

$e1 = 0.75, e2 = 0.2, e3 = 0.02, e4 = 0.02, e5 = 0.01$

This means that while predicting the word "राहुल", the decoder needs to put more attention on the states h1 and h2 (since values of e1 and e2 are high) while ignoring the states h3, h4 and h5 (since the values of e3, e4 and e5 are very small).

Step 3— Compute the context vector

Once we have computed the attention weights, we need to compute the context vector (thought vector) which will be used by the decoder in order to predict the next word in the sequence. Calculated as follows:

context_vector = e1 * h1 + e2 * h2 + e3 * h3 + e4 * h4 + e5 * h5

Clearly if the values of e1 and e2 are high and those of e3, e4 and e5 are low then the context vector will contain more information from the states h1 and h2 and relatively less information from the states h3, h4 and h5.

Step 4— Concatenate context vector with output of previous time step

Finally the decoder uses the below two input vectors to generate the next word in the sequence

a) The context vector

b) The output word generated from the previous time step.

We simply concatenate these two vectors and feed the merged vector to the decoder. **Note that for the first time step, since there is no output from the previous time step, we use a special <START> token for this purpose**. This concept is already discussed in detail in my previous *blog*.

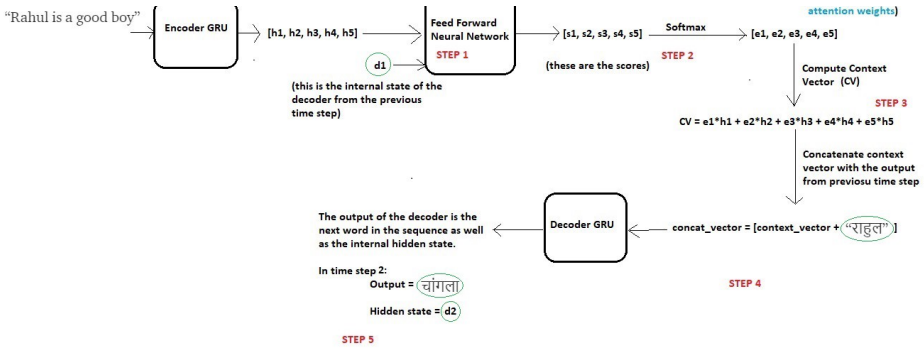
Step 5— Decoder Output

The decoder then generates the next word in the sequence (in this case, it is expected to generate "राहुल") and along with the output, the decoder will also generate an internal hidden state, and lets call it as "d1".

Decoding at time step 2

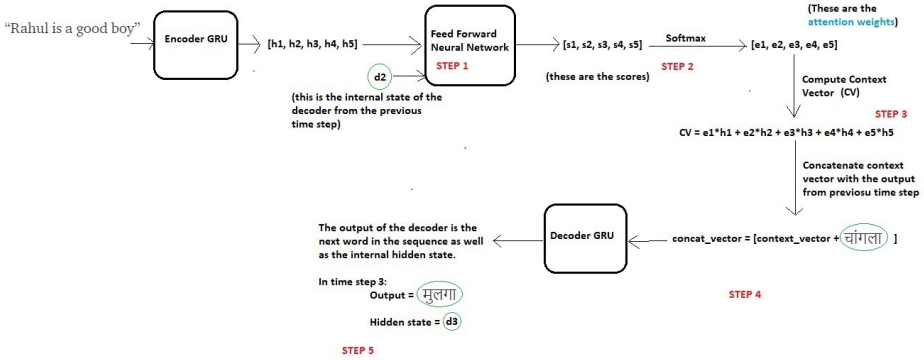
Now in order to generate the next word "चांगला", the decoder will repeat the same procedure which can be summarized in the below diagram:

The changes are highlighted in **green circles**



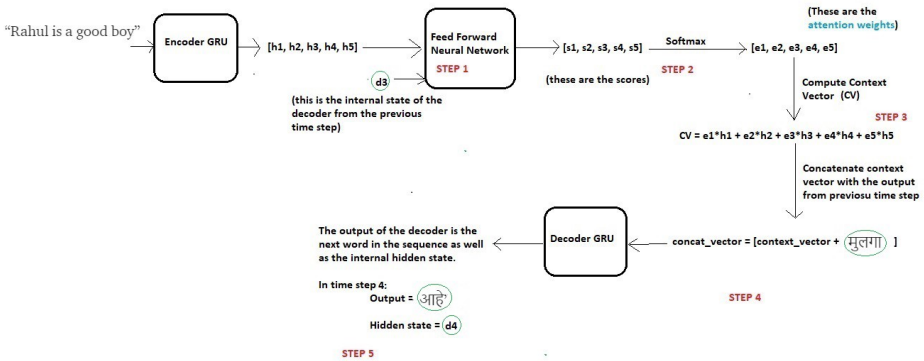
Decoding at time step 2

Decoding at time step 3



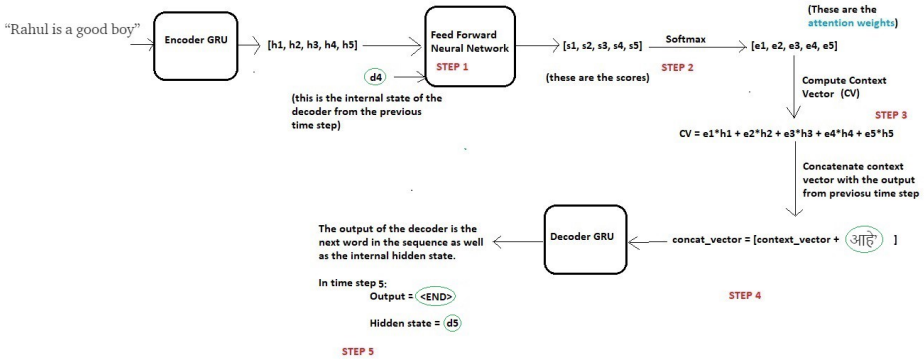
Decoding at time step 3

Decoding at time step 4



Decoding at time step 4

Decoding at time step 5



Once the decoder outputs the <END> token, we stop the generation process.

Note that unlike the fixed context vector used for all the decoder time steps in case of the traditional Seq2Seq models, here in case of Attention, we compute a separate context vector for each time step by computing the attention weights every time.

Thus using this mechanism our model is able to find interesting mappings between different parts of the input sequence and corresponding parts of the output sequence.

Note that during the training of the network, we use teacher forcing in order to input the actual word rather than the predicted word from the previous time step. This concept also has been explained in my previous *blog*.

5. Code Walk through

As in case of any NLP task, after reading the input file, we perform the basic cleaning and preprocessing as follows:

Create a class to map every word to an index and vice-versa for any given vocabulary:

We use the tf.data input pipeline to create the dataset and then load it later in mini batches. To read more about the input pipeline in TensorFlow, go through the official documentations *here* and *here*.

Now using the model sub-classing API of TensorFlow, we define the model as follows. To read more about model sub classing, read the official documentation *here*.

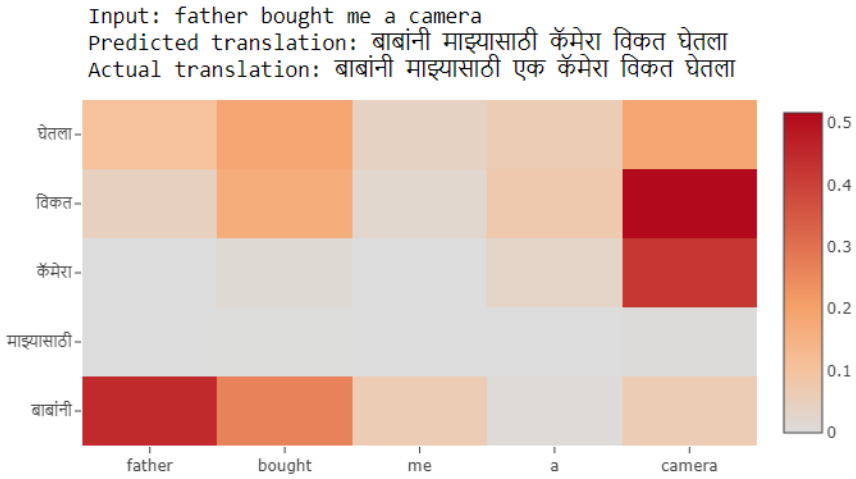
Note: Please read the comments in the below section of the code to get better understanding using the concepts we discussed above. Most of the important lines of the code point to the corresponding section of the explanation given above.

Define Optimizer, Loss Function and Checkpoints

Using Eager Execution, we train the network for 10 epochs. To read more about Eager Execution, refer the official documentation *here*.

Inference setup and testing:

6. Visualizing the Results

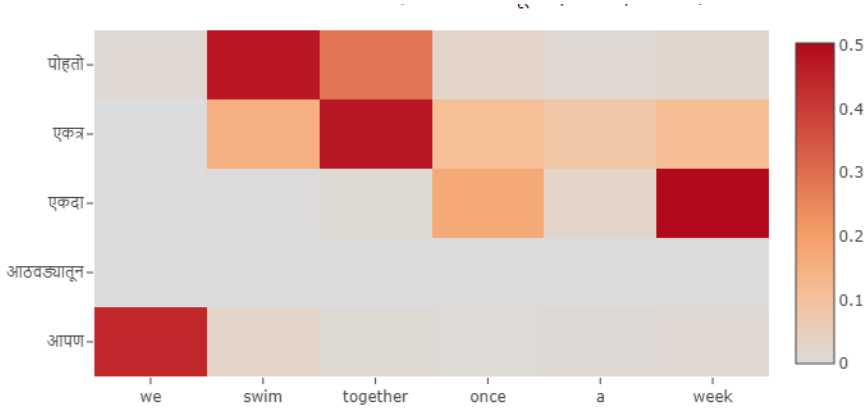


If you are new to heat maps, this is how you can interpret the above plot:

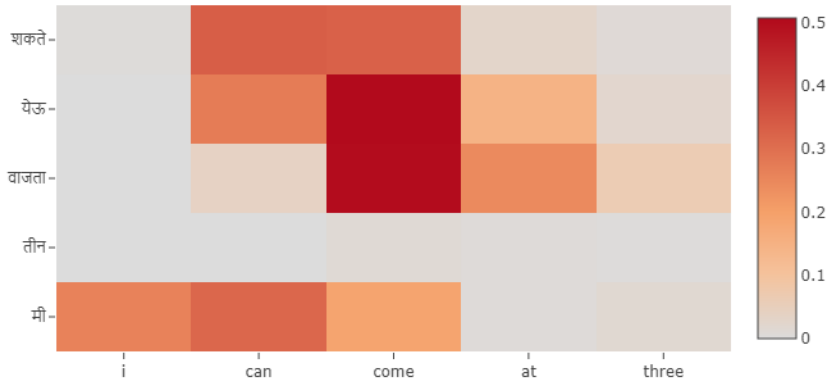
Notice that the cell at the intersection of “father” and “बाबांनी” is pretty dark This means when the decoder predicts the word “बाबांनी”, it is paying more attention to the input word “father” (which is what we wanted).

Similarly while predicting the word “कॅमेरा”, the decoder pays a lot of attention to the input word “camera”. And so on.

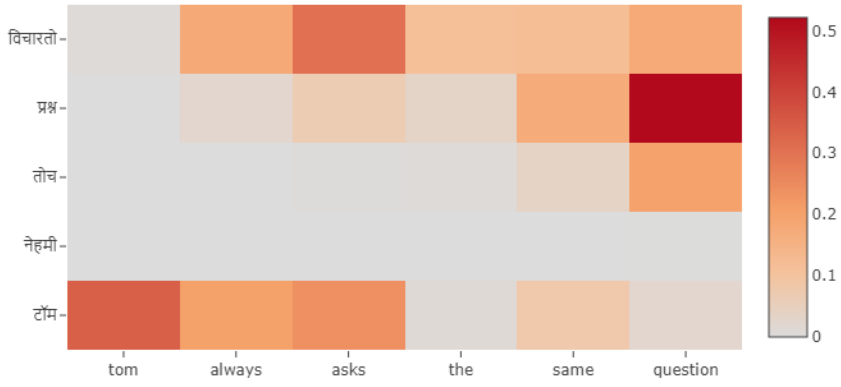
```
Input: we swim together once a week
Predicted translation: आपण आठवड्यातून एकदा एकत्र पोहतो
Actual translation: आम्ही आठवड्यातून एकदा एकत्र पोहतो
```



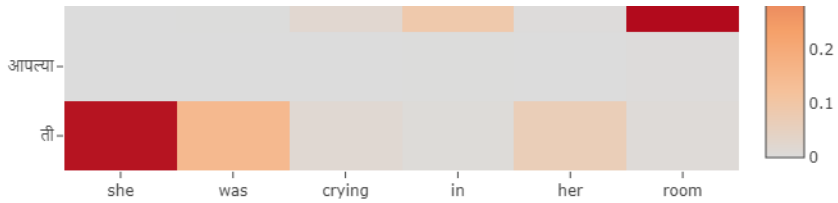
Input: i can come at three
Predicted translation: मी तीन वाजता येऊ शकते
Actual translation: मी तीन वाजता येऊ शकतो



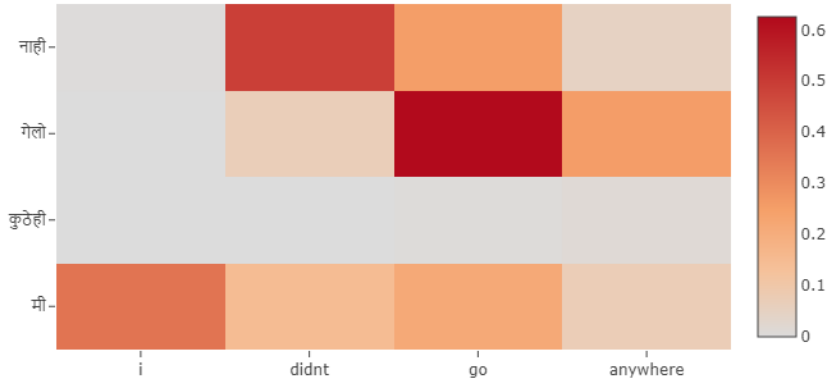
Input: tom always asks the same question
Predicted translation: टॉम नेहमी तोच प्रश्न विचारतो
Actual translation: टॉम नेहमी तो एकच प्रश्न विचारतो



Input: she was crying in her room
Predicted translation: ती आपल्या खोलीत रडत होती
Actual translation: त्या आपल्या खोलीत रडत होत्या




```
Input: i didnt go anywhere
Predicted translation: मी कुठेही गेलो नाही
Actual translation: मी कुठेही गेले नाही
```



Conclusion

The first thing to be noted is that the translation results are much better than the results from my previous *blog*. Secondly the model is able to find the correct local mappings between the input and the output sequences which do match with our intuition.

Given more data and with more hyper parameter tuning, the results and mappings will definitely improve by a good margin.

Using LSTM layers in place of GRU and adding Bidirectional wrapper on the encoder will also help in improved performance.

Deep Learning models are generally considered as black boxes, meaning that they do not have the ability to explain their outputs. However, Attention is one of the successful methods that helps to make our model interpretable and explain why it does what it does.

The only disadvantage of the Attention mechanism is that it is a very time consuming and hard to parallelize system. To solve this problem, Google Brain came up with the "Transformer Model" which uses only Attention and gets rid of all the Convolutional and Recurrent Layers, thus making it highly parallelizable and compute efficient.

7. References

- https://arxiv.org/abs/1409.0473 (Original Paper)
- https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/eager/python/examples/nmt_ Implementation available on their official website as a tutorial)
- https://www.coursera.org/lecture/nlp-sequence-models/attention-model-lSwVa(Andrew Ng's Explanation on Attention)
- https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

- https://www.tensorflow.org/xla/broadcasting (Broadcasting in TensorFlow)
- Dataset: http://www.manythings.org/anki/ (mar-eng.zip)

PS: For complete implementation, refer my GitHub repository *here*.