Neural Machine Translation using Transformers

1. INTRODUCTION

Recurrent Neural Networks are feed-forward neural nets rolled out over time. They deal with sequence data where the input has some defined ordering. This gives rise to several types of architectures:

1. *Vector to sequence Models*: This NN takes in a fixed-sized vector as input and it outputs a sequence of any length. In image captioning, the input can be a vector representation of an image and the output sequence is a sentence that describes the image.

2. *Sequence to vector models*: These NN takes in a sequence as input and outputs a fixed length of vector. In Sentiment Analysis, say a movie review is an input and a fixed size of the vector is the output indicating how good or bad this person thought this movie was.

3.*Sequence to Sequence Models*: These NN take in a sequence as input and outputs another sequence. In Language Translation, the input can be a sentence in English and the output is the translation in French (or any other language).

However, RNNs have some disadvantages. RNNs are slow. So slow that people used a 'truncated version of backpropagation' instead of doing a complete backpropagation step, in hopes to decrease the training time. But even using this truncated version of backpropagation was highly hardware intense. Another disadvantage of RNNs is that they cannot deal with Long Sequences. A basic RNN is shown in *Figure 1.1* and *Figure 1.2*.

Since then, there have been many advancements, and we saw LSTM (Long Short-Term Memory) cells in place of neurons. This LSTM cell allowed data scientists to deal with the problem of Long Sequences, however, these models still took forever to train.

For these RNNs and LSTM networks, input data needs to be passed sequentially (one after the other). We need inputs of the previous state to make any operations on the current state. This need to input data sequentially did not make use of GPUs very well which are designed for parallel computation.

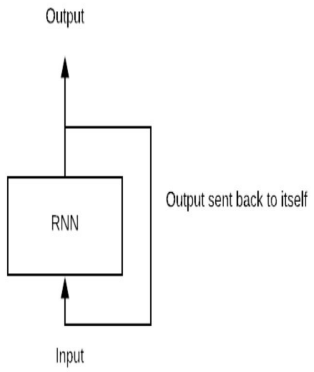


Figure 1.1

Question was how can we use parallelization for sequential data. The answer is **Transformers**.

Transformer NN architecture was introduced by Google in 2017 in their paper *'Attention is all you need'*. This used an encoder-decoder architecture, much like RNNs, but the difference was that unlike in RNNs, the input sequence can be passed in parallel.

With an *RNN encoder*, we had to pass an input English sentence one word after the other. The current word hidden states had dependencies in the previous word's hidden state. Basically, the word embeddings were generated one-time step at a time. Whereas, with a *Transformer*, there is no concept of the time step for the input. We pass in all the words of the sentence simultaneously and determine the word embeddings simultaneously.

Figure 1.2

Attention is All You Need

Transformer NN architecture was introduced by Google in 2017 in their paper *'Attention is all you need'*. Transformers try to solve the problem of parallelization by using a concept called 'Attention'.

How is Transformer better than RNN -

- No sequential input - Input is one whole sentence at a time (instead of one word at a time)
- Faster to train

2. PROJECT SCOPE

Our goal is to implement Language Translation using Transformer Neural Networks/Attention. We would be performing language translation from English to French for the ease of depiction.

3. ARCHITECTURE OVERVIEW

Transformer Neural Network consists of an Encoder(left) and a Decoder(right) block as shown in *Figure 3.1* below.

The Encoder takes an English sentence as input, and the output will be a set of encoded vectors called 'Attention Vectors' for every word in the English sentence. Each word in the input English sentence is converted into an embedding to represent meaning. We also add a positional vector to add the context of the word in the sentence. These word vectors are fed into the Encoder attention block (where the sentence goes through sublayer 1 and sublayer 2) which computes the attention vectors for every word.

The Decoder takes the following two things as input a) French word(s) b) Attention vectors from Encoder, and it generates the probabilities of the next French word. The French sentence is embedded for each word's meaning with the embedding layer and positional vectors are added to represent the context of the word in the sentence. These word vectors are fed into sublayer 1. Now, the output of sublayer 1 and the output from Encoder (Attention Vectors for English) are processed by sublayer 2 and 3 which produces attention vectors for every word in English-French mapping. These vectors are then passed into a linear layer and the softmax layer to predict the next French word. We repeat this process to generate the next word until the "end of sentence" token is generated.

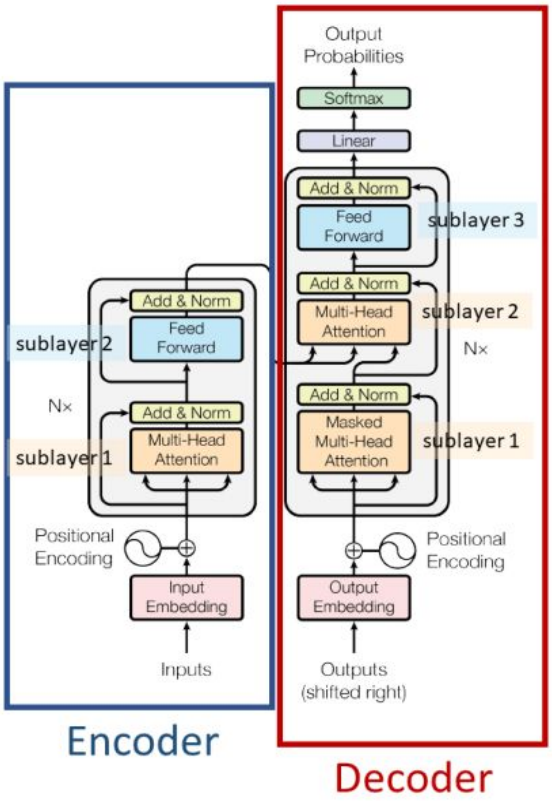


Figure 3.1

What we saw on the last page was the high-level details of how Transformer works. Let's dive deeper and examine each component.

3.1 Input / Output Embedding

Computers don't understand words, they understand numbers/vectors. The idea is to map every word to a point in high dimensional embedding space where similar words in meaning are physically closer to each other. *We can pre-train this embedding space to save time.*
This embedding space maps a word to a vector, but the same word in different sentences may have different meanings. This is where Positional Encoders come into picture.

3.2 Positional Encoder

The positional encoders receive inputs from the embeddings layer and apply relative positional information. This layer outputs word vectors with positional information; that is the word's meaning and its context in the sentence. An example is shown in *Figure 3.2.1*.

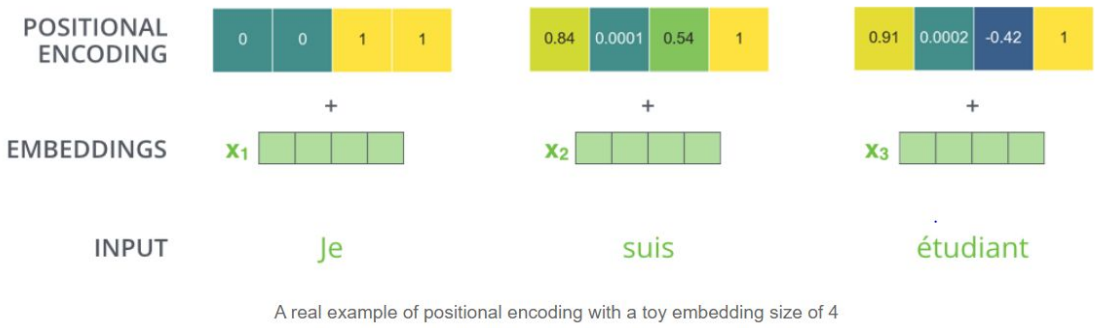


Figure 3.2.1

Consider the following sentences, "The dog bit Johnny" and "Johnny bit the dog." Without context information, both sentences would have almost identical embeddings. But we know this is not true.

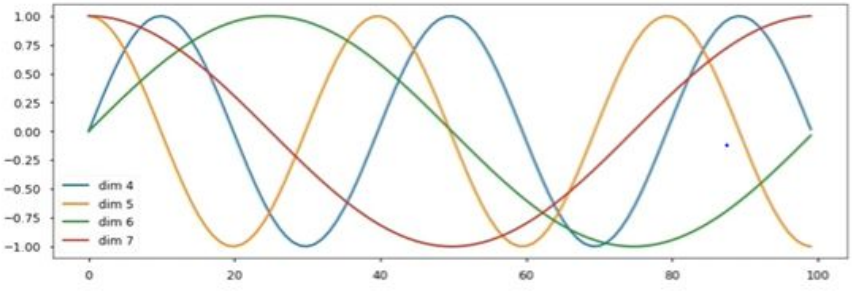
The authors proposed using multiple sine and cosine functions to generate positional vectors. This way, we can use this Positional Encoder for sentences of any length. The frequency and offset of the wave are different for each dimension, representing each position, with values between -1 and 1.

This binary encoding method also allows us to determine if two words are near each other. For example, by referencing the low-frequency sine wave, if one word has "high" while another is "low," we know that they are further apart, one located at the beginning, another at the end.

The idea is just to add something as a number that will be different for each position in the sentence and for each dimension of the embedding so that we lose the symmetry between each position.

So basically what we will do is that for each dimension of our embedding which is represented by the number, we will assign all the cosine functions with respect to the position in the sequence.

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/dmodel})$$
$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/dmodel})$$



pos: index of a word in a sentence (between 0 and seq_len - 1)

i: one of the dimensions of the embedding (between 0 and d_model - 1)

Figure 3.2.2

Let's look at *Figure 3.2.2*. If we take for instance the fourth dimension of all embeddings, if we go through all the positions in a sentence of length one hundred we can see that we have a certain sine function that has a certain frequency. But if we go to another dimension like dimension 6 we again get the sine function with respect to the position in the sequence but the frequency has changed.

So in all matrices representing our sequence we have a different number for each position in the sequence and each dimension in all embeddings.

3.3 Encoder Block

The encoder block consists of two layers

a. Attention Layer
b. Feed Forward Layer

a) Self-Attention/Multi-Head Attention Layer

The attention mechanism answers the question: What part of the input should we focus on?
For translation from English to French, we want to know how relevant a word in the english sentence is relevant to other words in the same sentence. This is represented in the attention vector.

For every word we can generate an attention vector that captures the contextual relationship between words in a sentence.

For example in *Figure 3.3.1*, for the word "black," the Attention mechanism focuses on "black" and "cat."

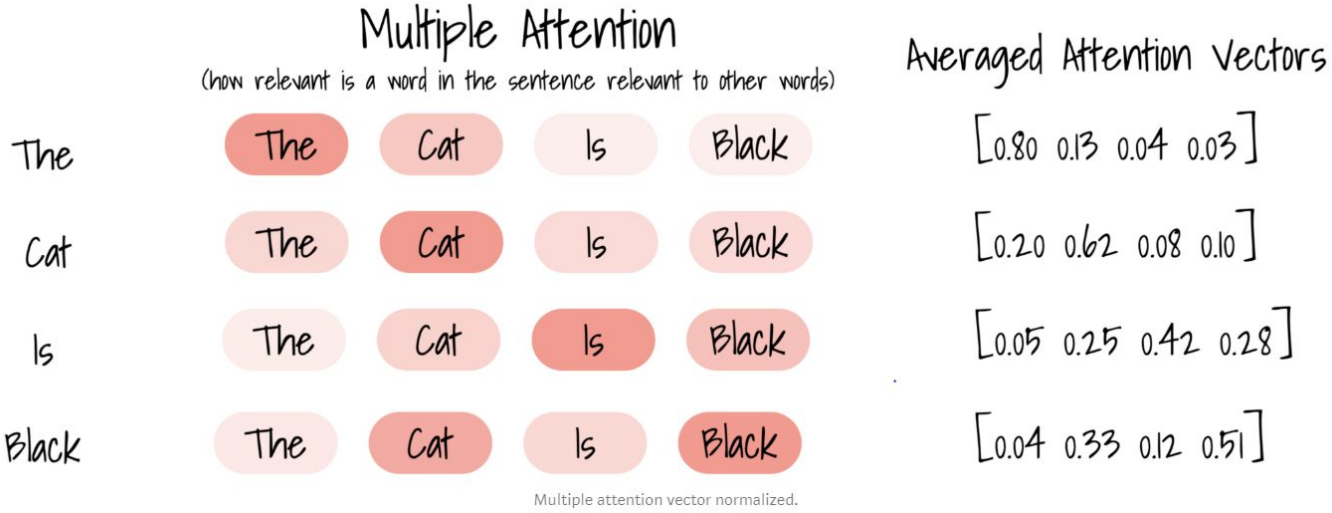


Figure 3.3.1

As we are interested in the interactions between different words, the Attention vector for each word may weigh itself too highly. As such, we need a way to normalize the vector.

The Attention layer accepts inputs V, K, and Q–these are abstract vectors that extract different components of an input word. We use these to compute the Attention vectors for every word.

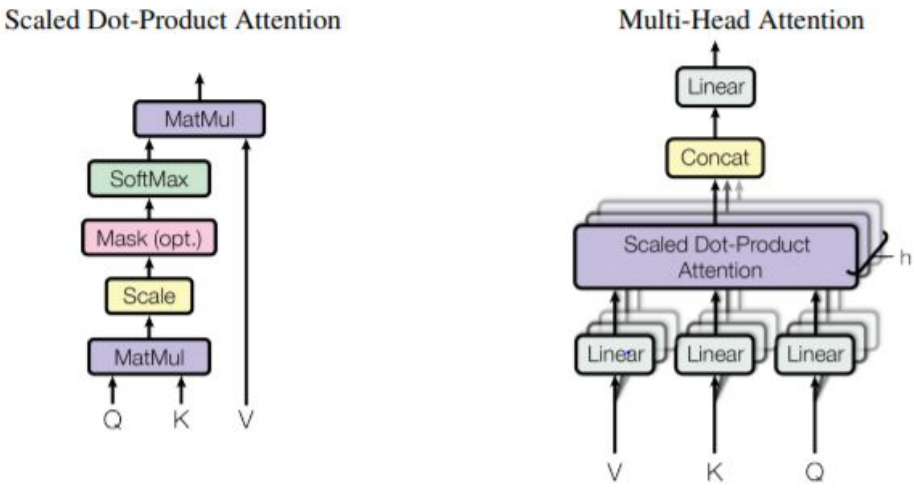


Figure 3.3.2

The equation used to calculate attention vectors is :

$$Attention(Q, K, V) = softmax_k(\frac{QK^T}{\sqrt{d_k}})V$$

Steps to calculate the attention vector

The **first step** in calculating self-attention is to create three vectors from each of the encoder's input vectors (in this case, the embedding of each word). So for each word, we create a Query vector, a Key vector, and a Value vector.

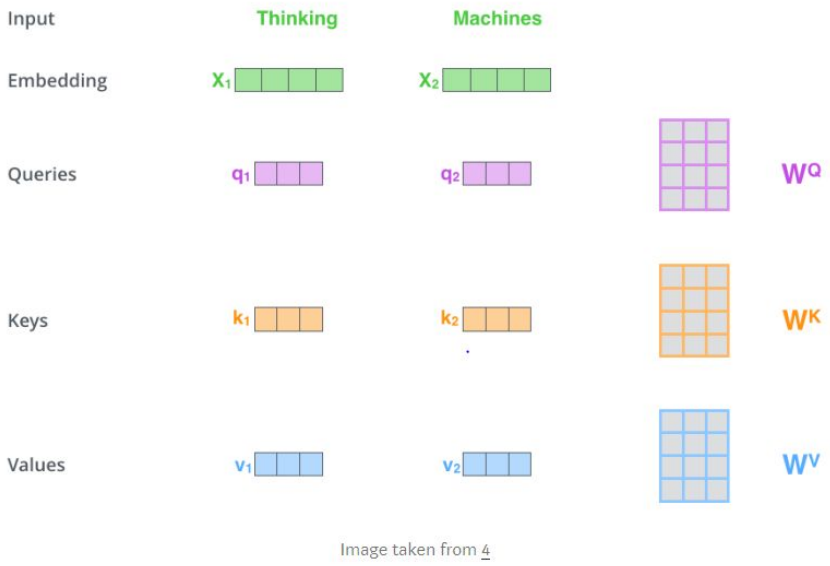


Figure 3.3.3

Figure 3.3.3 shows, multiplying x1 by the W^Q weight matrix produces q1, the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.

The **second step** in calculating self-attention is to calculate a score, shown in *Figure 3.3.4*

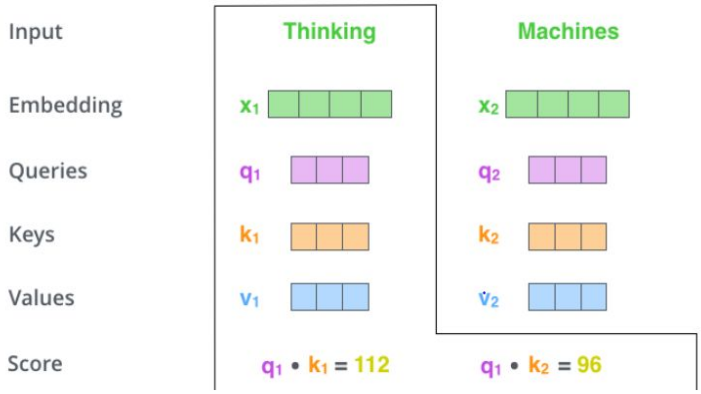


Figure 3.3.4

Say we're calculating the self-attention for the first word in this example, "Thinking". We need to score each word of the input sentence against this word. The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.
The score is calculated by taking the dot product of the query vector with the key vector of the respective word we're scoring. So if we're processing the self-attention for the word in position #1, the first score would be the dot product of q1 and k1. The second score would be the dot product of q1 and k2.

The **third and fourth steps** are to divide the scores by 8 *(the square root of the dimension of the key vectors used in the paper — 64. This leads to having more stable gradients)*, then pass the result through a softmax operation. Softmax normalizes the scores so they're all positive and add up to 1. This is shown in *Figure 3.3.5*

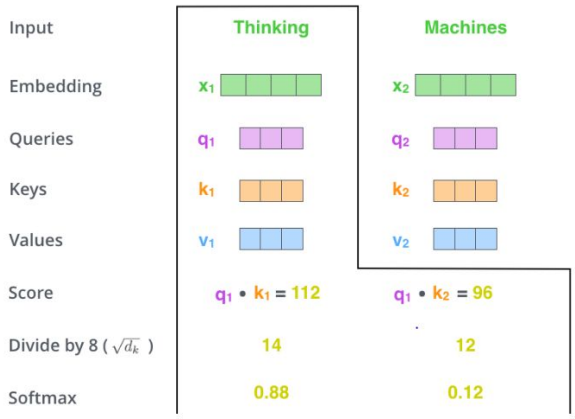


Figure 3.3.5

This softmax score determines how much each word will be expressed at this position. Clearly the word at this position will have the highest softmax score, but sometimes it's useful to attend to another word that is relevant to the current word.

The **fifth step** shown in *Figure 3.3.6*, is to multiply each value vector, v, by the softmax score (in preparation to sum them up). The intuition here is to keep intact the values of the word(s) we want to focus on and drown-out irrelevant words (by multiplying them by tiny numbers like 0.001, for example).

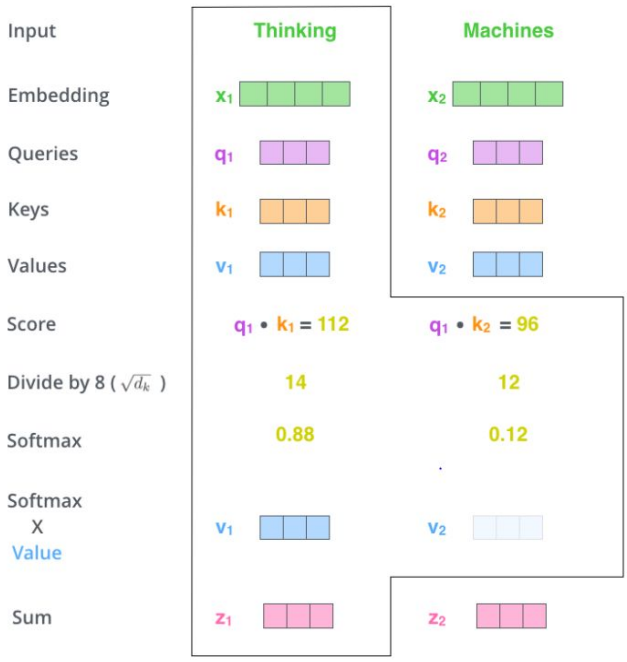


Figure 3.3.6

The **sixth step** shown in *Figure 3.3.7,* is to sum up the weighted value vectors. We do this because, as we saw in step 3 and 4, it is possible that the attention weight/softmax score of the word in a sentence with itself could be higher. Hence we perform this attention vector calculation multiple times and take a weighted average. This produces the output of the self-attention layer at this position (for the first word).

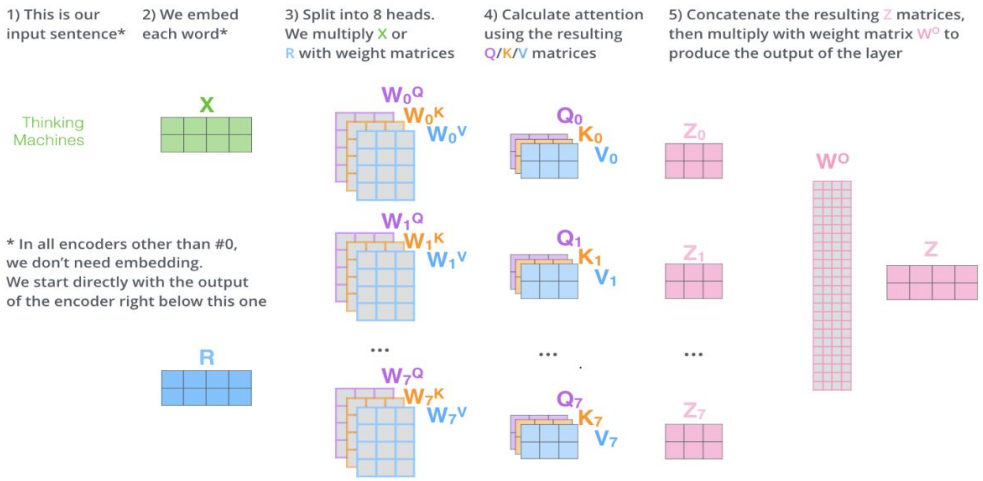


Figure 3.3.7

b) Feed Forward Layer

The attention vector computed in the previous step is the one we can send along to the feed-forward neural network. This is just a simple Feed Forward Neural Net that is applied to every one of the attention vectors. These FFN convert the attention vectors into a form which is digestible by the next encoder/decoder block.

Now you'll probably ask, RNNs are nothing but Feed Forward Nets, and we know RNNs are slow. So how is this any better?

Recall that RNNs were slow because current input was dependent on previous input. Well, the good part of these *Attention Vectors* (generated from the previous part) is that they are not dependent on each other, and thus each attention vector can be passed to Feed Forward Layer in parallel. Because of this, we can pass all our words at the same time into the encoder block and output will be a set of encoded vectors for every word. This is shown in *Figure 3.3.8*

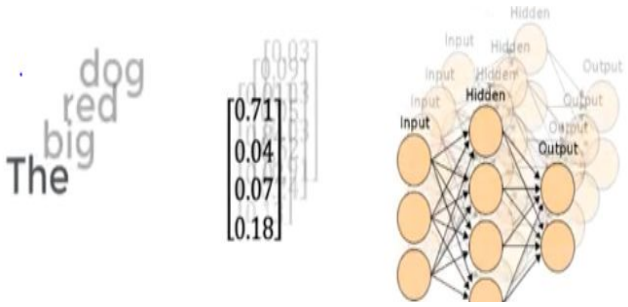


Figure 3.3.8

3.4 Decoder Block

The first input to the decoder is a French sentence that is embedded and positionally encoded like in the encoder section. The context vectors are then passed to the decoder block.

The decoder block consists of

a) Masked Attention Layer
b) Encoder Decoder Attention Layer - This is where the 2nd input is read.
c) Feed Forward Layer

a) Masked Attention Layer

Similar to the Multi-head Self-Attention layer in the encoder, the decoder's self attention block also generates attention vectors for every word in the French sentence to represent how relevant is each word to the other words in the sentence. The modification in this attention layer is that it includes a look-ahead mask.

The masked attention layer in the decoder stack prevents positions from attending to subsequent positions. This masking, combined with the fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i. In simpler words, While generating the next French word we can use all

words from the english sentence but only previous words from the French sentence. This is done by masking the future words for the decoder and the decoder would predict the next word of the french sentence. An example of this masking is shown in *Figure 3.4.1*

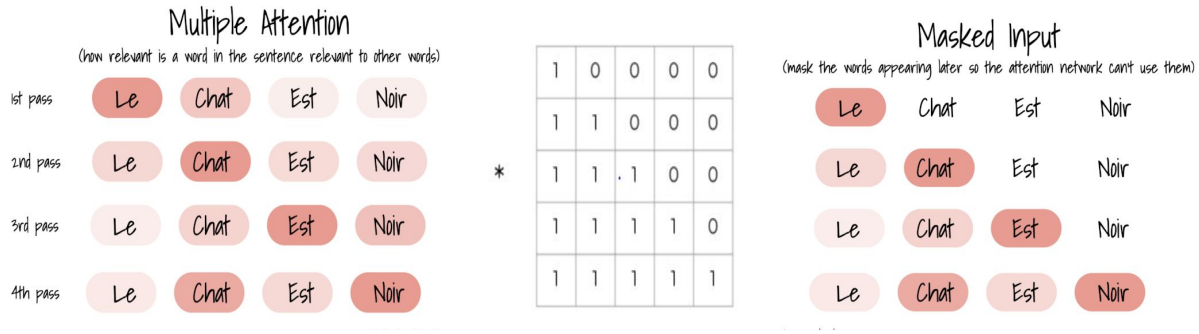


Figure 3.4.1

This masking is necessary to train the model to predict the next word in the sentence of the French translation. If all the words in the sentence are available to the decoder block, there would be no learning. Since our aim is to translate the English sentence to French, we want the decoder to learn to predict the words of the French sentence based on context.

b) Encoder Decoder Attention Layer

The *'Encoder-Decoder Attention'* layer works just like multiheaded self-attention, except it creates its Queries from the layer below it, and takes the Keys and Values from the output of the encoder. The attention vectors from Encoder are processed with attention vectors from sublayer 1 (French). This is where the English to French mapping happens. Shown in *Figure 3.4.2*

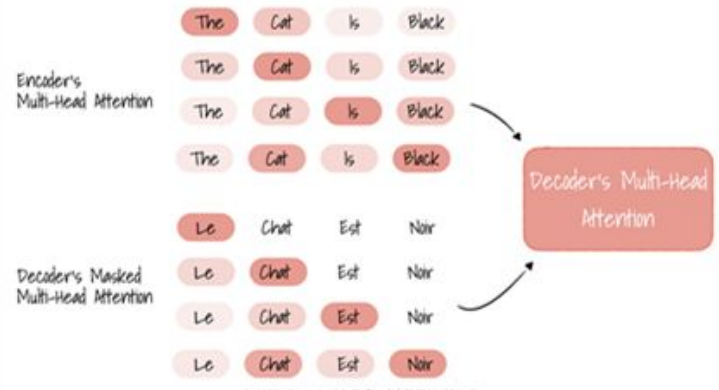


Figure 3.4.2

c) Feed Forward Layer

These FFN convert the attention vectors into a form which is digestible by the next linear layer. We have also seen FFNs are fast because they use Attention Vectors, which are not dependent on each other, and thus can be fed to FFNs in parallel.

3.5 Linear and Softmax Layer

The decoder layer outputs a vector of floating point numbers. How do we turn that into a word? That's the job of the final Linear layer which is followed by a Softmax Layer.

The Linear layer is a simple fully connected neural network that projects the vector produced by the stack of decoders, into a much, much larger vector called a logits vector.

Let's assume that our model knows 10,000 unique English words (our model's "output vocabulary") that it's learned from its training dataset. This would make the logits vector 10,000 cells wide – each cell corresponding to the score of a unique word. That is how we interpret the output of the model followed by the Linear layer.

The softmax layer then turns those scores into probabilities (all positive, all add up to 1.0). The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step. This is shown in *Figure 3.5*

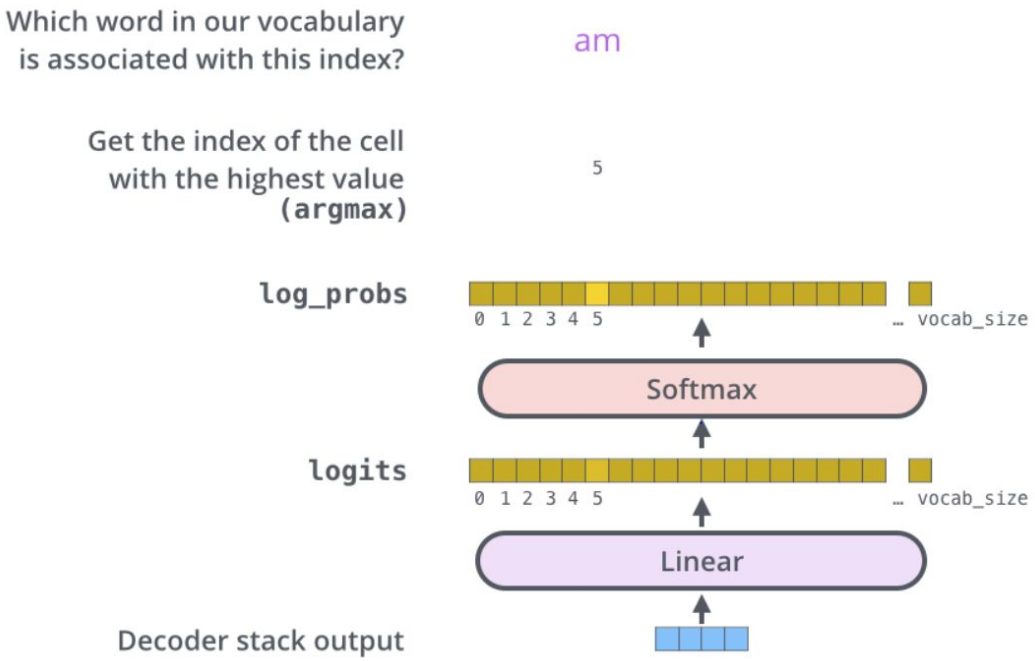


Figure 3.5

4. TRAINING

This section describes the training regime for our models.

4.1 Training Data and Batching

The training files can also be found through this link http://www.statmt.org/europarl/
For English-French, the data is extracted from the proceedings of the European Parliament. It includes versions in 21 European languages. The data is actually a parallel corpora of language pairs that include English.

Sentence pairs were batched together by approximate sequence length, max_length = 20. Each training batch contained a set of sentence pairs containing approximately 25000 source tokens and 25000 target tokens.

4.2 Hardware and Schedule

We implemented our project in Google Colab as it is a free online cloud-based Jupyter notebook environment that allows us to train our machine learning and deep learning models on CPUs, GPUs, and TPUs. For our model, using the hyperparameters described throughout the paper, each training step took about 0.4 seconds. We trained the base models for a total of 100,000 steps or 12 hours.

4.3 Optimizer

We used the Adam optimizer with $\beta 1 = 0.9$, $\beta 2 = 0.98$ and epsilon = 10^{-9}

We varied the learning rate over the course of training, according to the formula:

$$lrate = d{-}0.5model{*}min(stepNum{-}0.5, stepNum{*}warmupSteps{-}1.5)$$

The research paper uses a custom learning rate. So we created a class which follows the Learning Rate as defined above. This corresponds to increasing the learning rate linearly for the first warmup_steps training steps, and decreasing it thereafter proportionally to the inverse square root of the step number. We used warmup_steps = 4000.

4.4 Regularization

We employ three types of regularization during training:

- Residual Dropout
- Dropout to the output of each sub-layer, before it is added to the sub-layer input and normalized.
- Dropout to the sums of the embeddings and the positional encodings in both the encoder and decoder. For the base model, we use a rate of dropout_rate=0.1.

4.5 Loss Function

The output of our decoder is a probability value, predicting the next words in the French sequence, so our loss object would be sparse categorical cross entropy.

5. HOW TO RUN OUR TRANSLATOR?

The code we used to train and evaluate Eng-French can be found here. You can also access other translator files like Eng-Spanish from here. After you have the access to google colab file follow these steps mentioned below.

Please follow the following steps to run the code:

1. Open Google Colab Notebook from the link provided above.

2. Download the data and checkpoint/pickle files from this link and add it to your drive. (Remember if you are uploading our files directly on your drive, you do not need to change any directory paths. If you are uploading our file/folders inside some directory on your drive, please change the directory address in the code accordingly while mounting).

3. Run the code file. On code cell 2 (shown below in image) you will be asked to mount your Google Drive. You can mount by clicking on the link given in code. Copy paste the authorization code in the same code cell and press enter. (You should mount the drive where you uploaded our data files).

4. Simply run the remaining code cells to use the translator.

5. OPTIONAL: If you wish to retrain the model, you can do so by uncommenting the 'training' code cell. However, keep in mind, each epoch takes around 2 hours to train.

6. CONCLUSION

In this work, we presented the Transformer, the first sequence transduction model based entirely on attention, replacing the slow recurrent layers with multi-headed self-attention, following the paper " Attention is all we need".

For translation tasks, the Transformer can be trained significantly faster than architectures based on recurrent or convolutional layers.

On running the code, you will get results similar to *Figure 6.1*

Figure 6.1

References

- https://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf
- https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html
- https://towardsdatascience.com/illustrated-guide-to-transformer-cf6969ffa067