Eduardo Muñoz · Follow
Nov 2, 2020 · 13 min read · Listen

GETTING STARTED

Attention is all you need: Discovering the Transformer paper

Detailed implementation of a Transformer model in Tensorflow

Picture by Vinson Tan from Pixabay

In this post we will describe and demystify the relevant artifacts in the paper *"Attention is all you need" (Vaswani, Ashish & Shazeer, Noam & Parmar, Niki & Uszkoreit, Jakob & Jones, Llion & Gomez, Aidan & Kaiser, Lukasz & Polosukhin, Illia. (2017))[1]*. This paper was a great advance in the use of the attention mechanism, being the main improvement for a model called Transformer. The most famous current models that are emerging in NLP tasks consist of dozens of transformers or some of their variants, for example, GPT-2 or BERT.

We will describe the components of this model, analyze their operation and build a simple model that we will apply to a small-scale NMT problem (Neural Machine Translation). To read more about the problem that we will address and to know how the basic attention mechanism works, I recommend you to read my previous post "A Guide on the Encoder-Decoder Model and the Attention Mechanism".

Why we need Transformer

In sequence-to-sequence problems such as the neural machine translation, the initial proposals were based on the use of RNNs in an encoder-decoder architecture. These architectures have a great limitation when working with long sequences, their ability to retain information from the first elements was lost when new elements were incorporated into the sequence. In the encoder, the hidden state in every step is associated with a certain word in the input sentence, usually one of the most recent. Therefore, if the decoder only accesses the last hidden state of the decoder, it will lose relevant information about the first elements of the sequence. Then to deal with this limitation, a new concept were introduced **the attention mechanism**.

Instead of paying attention to the last state of the encoder as is usually done with RNNs, in each step of the decoder we look at all the states of the encoder, being able to access information about all the elements of the input sequence. This is what attention does, it extracts information from the whole sequence, a **weighted sum of all the past encoder states**. This allows the decoder to assign greater weight or importance to a certain element of the input for each element of the output. Learning in every step to focus in the right element of the input to predict the next output element.

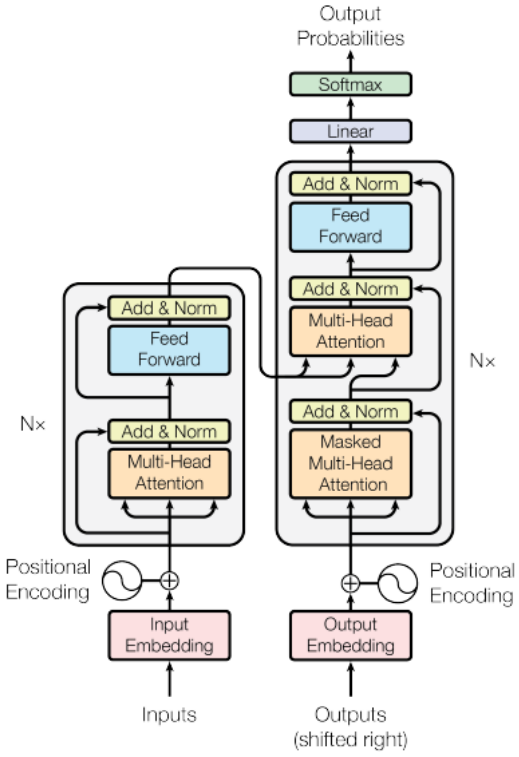
time consuming and computationally inefficient.

What is the Transformer?

> *In this work we propose the Transformer, a model architecture eschewing recurrence and instead relying entirely on an attention mechanism to draw global dependencies between input and output. The Transformer allows for significantly more parallelization … the Transformer is the first transduction model relying entirely on self-attention to compute representations of its input and output without using sequence-aligned RNNs or convolution.*
>
> *"Attention is all you need" paper [1]*

The Transformer model extract features for each word using a self-attention mechanism to **figure out how important all the other words in the sentence are** w.r.t. to the aforementioned word. And no recurrent units are used to obtain this features, they are just weighted sums and activations, so they can be very parallelizable and efficient.

But we will dive deeper into its architecture (next figure) to understand what all this pieces do [1].



From "Attention is all you need" paper by Vaswani, et al., 2017 [1]

We can observe there is an encoder model on the left side and the decoder on the right one. Both contains a core block of "an attention and a feed-forward network" repeated N times. But first we need to explore a core concept in depth: the self-attention mechanism.

Self-Attention: the fundamental operation

> *Self-attention is a sequence-to-sequence operation: a sequence of vectors goes in, and a sequence of vectors comes out. Let's call the input vectors* $x_1, x_2, \ldots, x_t$ *and the corresponding output vectors* $y_1, y_2, \ldots, y_t$. *The vectors all have dimension k. To produce output vector* y_i, *the self attention operation simply takes* a weighted average over all the input vectors, *the simplest option is the dot product.*
>
> *Transformers from scratch by Peter Bloem [2]*

In the self-attention mechanism of our model we need to introduce three elements: Queries, Values and Keys

once the weights have been established (Value).

To obtain this roles, we need three weight matrices of dimensions *k x k* and compute three linear transformation for each `xi` :

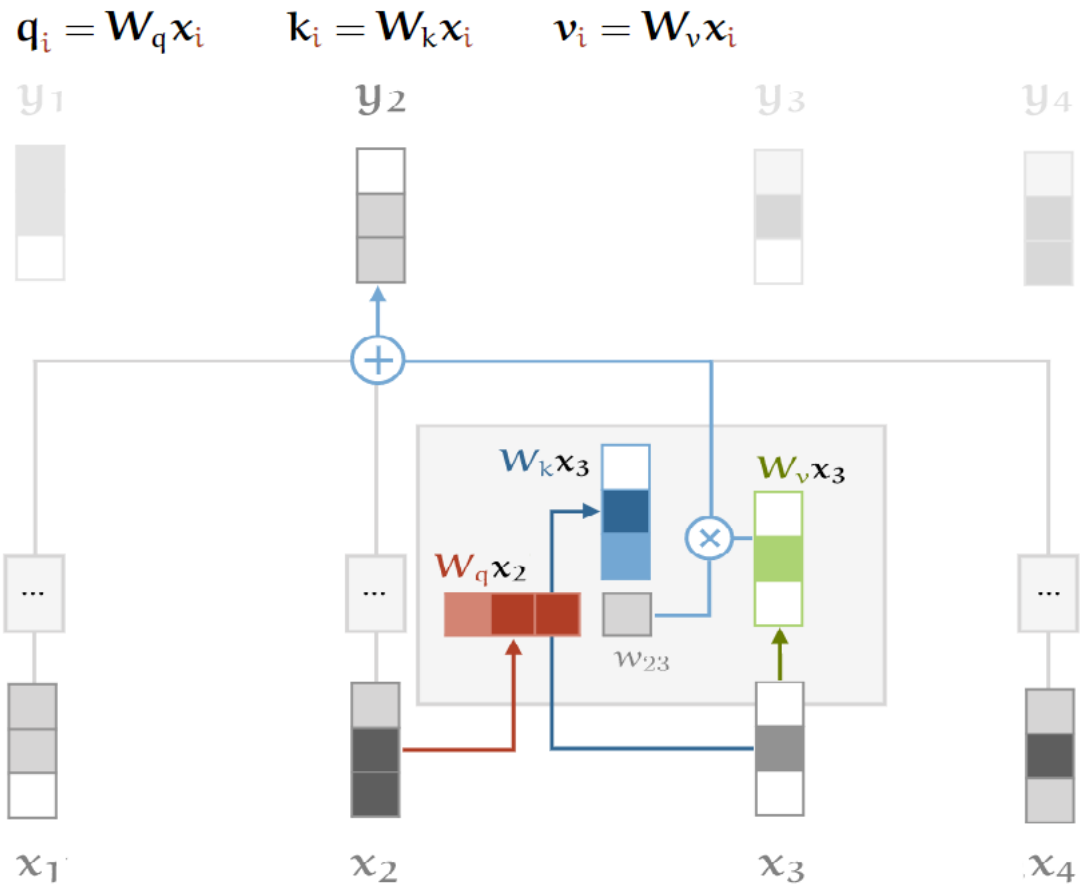


Illustration of the self-attention with key, query and value

"Transformers from scratch" by Peter Bloem [2]

These three matrices are usually known as *K*, *Q* and *V,* **three learnable weight layers that are applied to the same encoded input**. Consequently, as each of these three matrices come from the same input, we can apply the attention mechanism of the input vector with itself, a "self-attention".

The Scaled Dot-Product Attention

> *The input consists of queries and keys of dimension* `dk` *, and values of dimension* `dv` *. We compute the dot product of the query with all keys, divide each by the square root of* `dk` *, and apply a* `softmax` *function to obtain the weights on the values.*
>
> *"Attention is all you need" paper [1]*

Then we use the *Q*, *K* and *V* matrices to calculate the attention scores. **The scores measure how much focus to place on other places or words of the input sequence w.r.t a word at a certain position**. That is, the dot product of the query vector with the key vector of the respective word we're scoring. So, for position 1 we calculate the dot product (.) of `q1` and `k1` , then `q1 . k2` , `q1 . k3` and so on,…

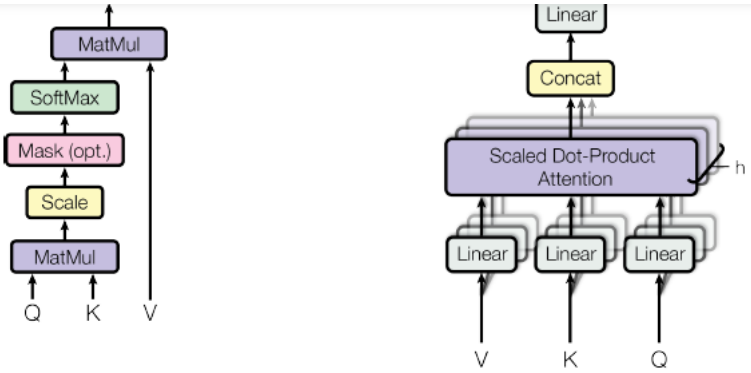
Next we apply the "scaled" factor to have more stable gradients. The softmax function can not work properly with large values, resulting in vanishing the gradients and slowing down the learning, [2]. After "softmaxing" we multiply by the Value matrix to keep the values of the words we want to focus on and minimizing or removing the values for the irrelevant words (its value in V matrix should be very small).

$\sqrt{d_k}$

From “Attention is all you need” paper by Vaswani, et al., 2017 [1]. Scaled dot-product attention formula.

Multi-head Attention

In the previous description the attention scores are focused on the whole sentence at a time, this would produce the same results even if two sentences contain the same words in a different order. Instead, we would like to attend to different segments of the words. *“We can give the self attention greater power of discrimination,* ***by combining several self attention heads,*** *dividing the words vectors into a fixed number (h, number of heads) of chunks, and then self-attention is applied on the corresponding chunks, using Q, K and V sub-matrices.”*, [2] Peter Bloem, “Transformers from scratch”. This produce *h* different output matrices of scores.



From “Attention is all you need” paper by Vaswani, et al., 2017 [1]

But the next layer (the Feed-Forward layer) is expecting just one matrix, a vector for each word, so *“after calculating the dot product of every head, we concatenate the output matrices and multiply them by an additional weights matrix* W^O*,”*[3]. This final matrix captures information from all the attention heads.

Positional Encoding

We mentioned briefly that the order of the words in the sentence is an issue to solve in this model, because the network and the self-attention mechanism is permutation invariant. If we shuffle up the words in the input sentence, we get the same solutions. We need to create a representation of the position of the word in the sentence and add it to the word embedding.

> *To this end, we add "positional encodings" to the input embeddings at the bottoms of the encoder and decoder stacks. The positional encodings have the same dimension as the embeddings, so that the two can be summed. There are many choices of positional encodings.*
>
> *"Attention is all you need" paper*

So, we apply a function to map the position in the sentence to a real valued vector. The network will learn how to use this information. Another approach would be to use a position embedding, similar to word embedding, coding every known position with a vector. *"It would requiere sentences of all accepted positions during the training loop but positional encoding allow the model to extrapolate to sequence lengths longer than the ones encountered during training"*, [2].

In the paper a sinusoidal function is applied:

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

From "Attention is all you need" paper by Vaswani, et al., 2017 [1]. Positional encoding

The encoder

Now that all the main pieces of the model have been described we can introduce the encoder components, [4]:

- **Positional encoding**: Add the position encoding to the input embedding (our input words are transformed to embedding vectors). *"The same weight matrix is shared between the two embedding layers (encoder and decoder) and the pre-softmax linear transformation. In the embedding layers, we multiply those weights by square root of the model dimension"* [1].
- `N=6` identical layers, containing two sub-layers: a **multi-head self-attention** mechanism, and a **fully connected feed-forward network (**two linear transformations with a ReLU activation). But it is applied position-wise to the input, which means that the same neural network is applied to every single "token" vector belonging to the sentence sequence.

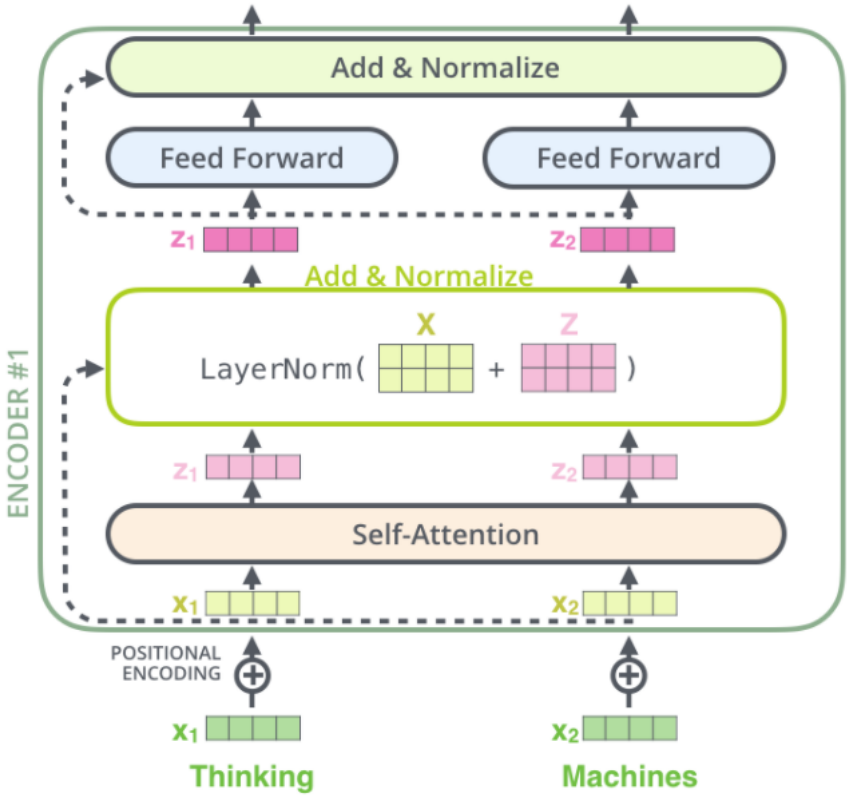
$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

- There is a **residual connection** around each sub-layer (attention and FC network), summing up the output of the layer with its input, followed by a **layer normalization**.
- Before every residual connection, a **regularization** is applied: *"We apply dropout to the output of each sub-layer, before it is added to the sub-layer input and normalized. In addition, we apply dropout to the sums of the embeddings and the positional encodings in both the encoder and decoder stacks"* [1] with a dropout rate of 0.1.

> Normalization and residual connections are standard tricks used to help deep neural networks train faster and more accurately. The layer normalization is applied over the embedding dimension only.
>
> Peter Bloem, "Transformers from scratch" [2]

First we implement the encoder layer, each one of the six blocks, contained in an encoder:

The next figure will show the components detailed:



"The Ilustrated Transformer" by Jay Alammar [3]

And the encoder code:

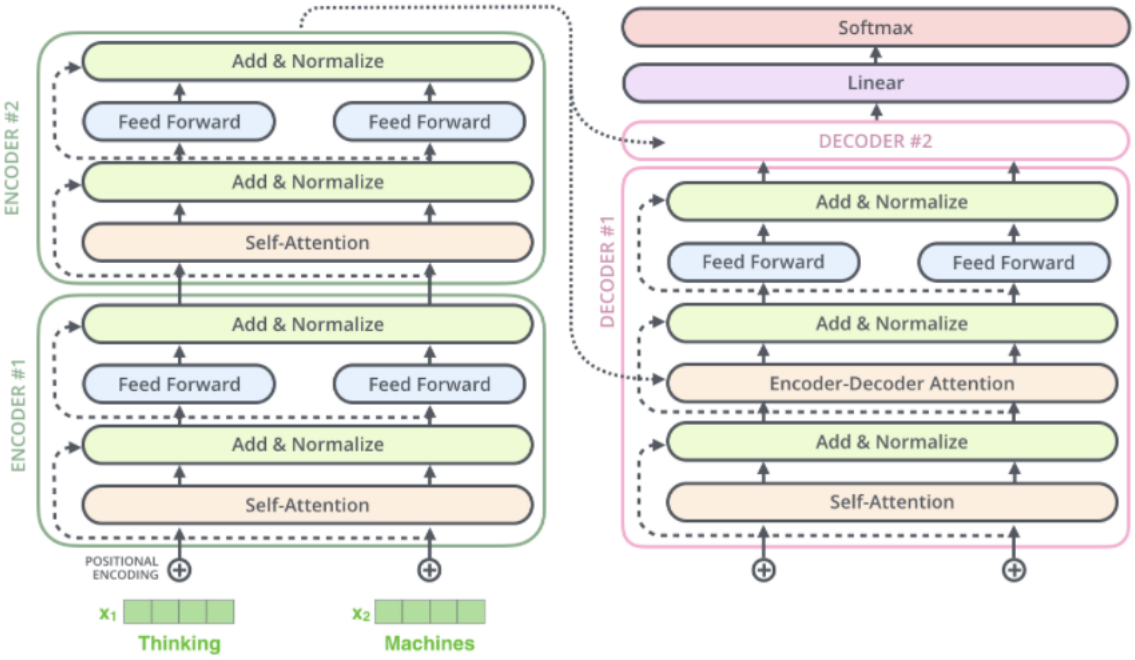
Keep in mind that **only the vector from the last layer (6-th) is sent to the decoder**.

The Decoder

The decoder share some components with the encoder but they are used in a different way to take into account the encoder output, [4]:

- **Positional encoding**: Similar that the one in the encoder
- `N=6` identical layers, containing 3 three sub-layers. First, the Masked Multi-head attention or **masked causal attention** to prevent positions from attending to subsequent positions. *"This masking, combined with 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" [1]*. It is implemented setting to $-\infty$ the values corresponding to the forbidden states in the softmax layer of the dot-product attention modules. The **second component** or **"encoder-decoder attention"** performs multi-head attention over the output of the decoder, the Key and Value vectors come from the output of the encoder but the queries come from the previous decoder layer. *"This allows every position in the decoder to attend over all positions in the input sequence" [1]*. And finally the fully-connected network.
- The **residual connection** and **layer normalization** around each sub-layer, similar to the encoder.
- And repeat the same **residual dropout** that was executed in the encoder.

The decoder layer:



"The Ilustrated Transformer" by Jay Alammar [3]

At the end of the N stacked decoders, the **linear layer**, a fully-connected network, transforms the stacked outputs to a much larger vector, *the logits*. *"The **softmax layer** then turns those scores (logits) into probabilities (all positive, all add up to 1.0). The cell with the*

Joining all the pieces: the Transformer

Once we have defined our components and created the encoder, the decoder and the linear-softmax final layer, we join the pieces to form our model, the Transformer.

It is worth mentioning that **we create 3 masks**, each of which will allow us:

- *Encoder mask*: It is a padding mask to discard the pad tokens from the attention calculation.
- *Decoder mask 1*: this mask is a union of the padding mask and the look ahead mask which will help the causal attention to discard the tokens "in the future". We take the maximum value between the padding mask and the look ahead one.
- *Decoder mask* 2: it is the padding mask and is applied in the encoder-decoder attention layer.

As you can see, then we call the encoder, the decoder and the final linear-softmax layer to get the predicted output from our Transformer model.

a NMT problem. It is a toy problem for educational purposes.

We won't deal with the data wrangling in this blog post. Follow the link I mentioned in the introduction for more information and the code provided to see how the data is loaded and prepared. In summary, create the vocabulary, tokenize (including a `eos` and `sos` token) and pad the sentences. Then we create a Dataset, a batch data generator, for training on batches.

We need to **create a custom loss function** to mask the padding tokens.

We use an Adam optimizer described in the paper, with `beta1=0.9`, `beta2=0.98` and `epsilon=10e-9`. And then we **create a scheduler to vary the learning rate** over the training process according to:

$$lrate = d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$$

"Attention is all you need" paper. Learning rate decay.

The main train function

The train function is similar to many other Tensorflow trainings, an usual training loop for sequence-to-sequence tasks:

- For every iteration on the batch generator that produce batch size inputs and outputs
- Get the input sequence from 0 to length-1 and the actual outputs from 1 to length, the next word expected at every sequence step.
- Call the transformer to get the predictions
- Calculate the loss function between the real outputs and the predictions
- Apply the gradients to update the weights in the model and update the optimizer too
- Calculate the mean loss and the accuracy for the batch data
- Show some results and save the model in every epoch

And that's all, we have all the necessary elements to train our model, we just need to create them and call the train function:

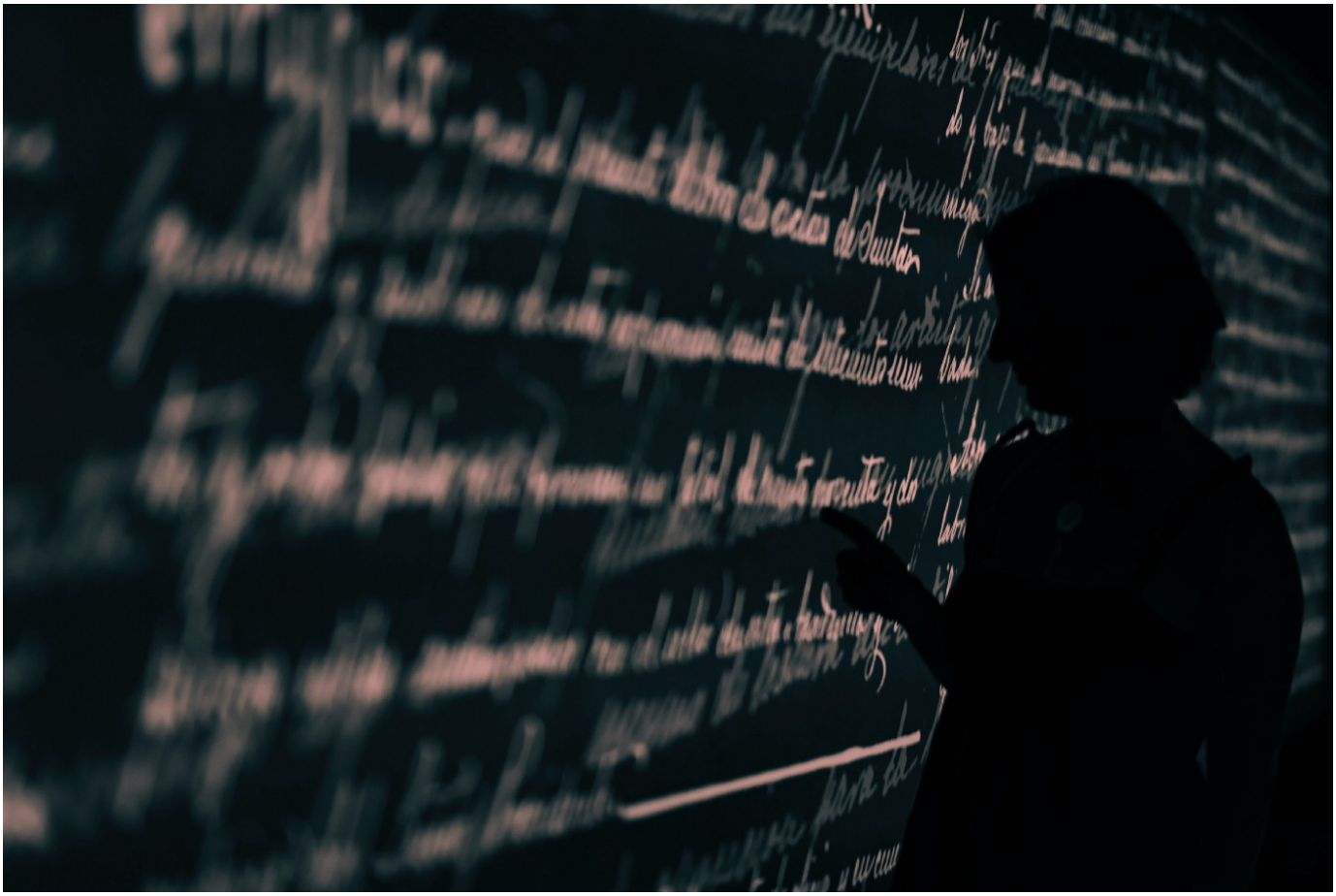


Photo by Jr Korpa on Unsplash

Make predictions

When training a ML model we are not only interested in optimize losses or accuracies, we want our model to make good enough predictions and, in this case, see how the model works with new sentences. The **predict function** will input a tokenized sentence to the model and return the predicted new sentence, in our example, a translation from English to Spanish.

These are the steps in that process:

- Tokenize the input sentence to a sequence of tokens

- Get the next word predicted. The model returns the logits, remember that the softmax function is applied in the loss calculation.
- Get the index in the vocabulary of the word with the highest probability
- Concatenate the next word predicted to the output sequence

And finally our last function receives a sentence in English, calls the transformer to translate it to Spanish and shows the result.

For this example, we just experiment with some values for the model dimension and the units of the feedforward network to train the model for an hour. If you want to optimize the model, you should probably train it for longer and with many different values for the hyperparameters.

The code is available in my github repository "Transformer-NMT". The code is partially extracted from an excellent course by SuperDataScience Team called "Modern Natural Language Processing in Python" on Udemy. I highly recommend it.

Some examples of translations are:

```
#Show some translations
sentence = "you should pay for it."
print("Input sentence: {}".format(sentence))
predicted_sentence = translate(sentence)
print("Output sentence: {}".format(predicted_sentence))

Input sentence: you should pay for it.
Output sentence: Deberías pagar por ello.

#Show some translations
sentence = "we have no extra money."
print("Input sentence: {}".format(sentence))
predicted_sentence = translate(sentence)
print("Output sentence: {}".format(predicted_sentence))

Input sentence: we have no extra money.
Output sentence: No tenemos dinero extra.

#Show some translations
sentence = "This is a problem to deal with."
print("Input sentence: {}".format(sentence))
predicted_sentence = translate(sentence)
print("Output sentence: {}".format(predicted_sentence))
```

References

[1] Vaswani, Ashish & Shazeer, Noam & Parmar, Niki & Uszkoreit, Jakob & Jones, Llion & Gomez, Aidan & Kaiser, Lukasz & Polosukhin, Illia, "Attention is all you need" , 2017.

[2] Peter Bloem, "Transformers from scratch" blog post, 2019.

[3] Jay Alammar, "The Ilustrated Transformer" blog post, 2018.

[4] Lilian Weng, "Attention? Attention!!" blog post, 2018.

[5] Ricardo Faúndez-CArrasco, "Attention is all you need's review" blog post, 2017

[6] Alexander Rush, "The Annotated Transformer", 2018, Harvard NLP group.