Tom Cordruwisch
R504D80 Deep Learning

15.05.2024

Exercise project 5 - Transformer networks

To confirm and improve my understanding of this topic, I made use of ChatGPT by asking questions in order to be able to match up my own comprehension with the resources I studied on the internet, but I did not include any of the generated text in this document.

To start, I followed along with a basic implementation of a transformer neural network for text translation.
https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/nlp/ipynb/neural_machine_translation_with_transformer.ipynb

I first made necessary instals and imports, downloaded the dataset with example sentence pairs, provided in English and with their respective Spanish translation.
The sentences were converted into a fitting form to serve as the dataset to train the transformer. This included test-train split, stripping of special characters and vectorising the sentences word for word.
The transformer model consists of multiple connected layers: an encoder layer handling the input sentences (English) and a decoder layer for the output sentences. (Spanish translations)
Word order in the sentences is embedded into the data as it is important information for transformers.
I will look into the structure in more detail later, but for now I can see that the model contains all the major parts of a transformer: decoder and encoder layers, positional embedding and connections between them:

Model: "transformer"

Layer (type)	Output Shape	Param #	Connected to
encoder_inputs (InputLayer)	(None, None)	0	-
positional_embedding (PositionalEmbedding)	(None, None, 256)	6,405,120	encoder_inputs[0][0]
decoder_inputs (InputLayer)	(None, None)	0	-
transformer_encoder (TransformerEncoder)	(None, None, 256)	3,155,456	positional_embedding[…
functional_5 (Functional)	(None, None, 25000)	18,089,640	decoder_inputs[0][0], transformer_encoder[0…

It’s worth noting that the functional layer at the bottom is where the decoder and encoders connect.

It was mentioned that for convergence, where the model approximately achieves the best metrics it can with the particular configuration, the training should be 30 epochs or more.
Because I didn’t require convergence for my purpose, but just wanted to test a model that functions (even if the translations are just passable), I trained for 7 epochs. The training was completed after 10 minutes with a final validation accuracy of about 88%, although it is stated that the BLEU scores and other metrics are more commonly used to assess a transformer model.

After the training, I was able to test some example translations. Original sentence and ML-translation in alternating order:

```
Original sentence: Do what you like.
Translation: [start] haz lo que quieras [end]

Original sentence: I always knew Tom was an idiot.
Translation: [start] siempre sabía que tom era un idiota [end]

Original sentence: That's what happens when you lend things to people.
Translation: [start] eso es lo que pase las cosas a las cosas [end]

Original sentence: This house is very big, isn't it?
Translation: [start] esta casa es muy grande no [end]

Original sentence: Tom likes the idea.
Translation: [start] a tom le gusta la idea [end]

Original sentence: Tom has been fired.
Translation: [start] tom ha sido despedido [end]

Original sentence: I do not agree.
Translation: [start] yo no estoy de acuerdo [end]
```

There definitely were some bad translations, but at least half of them seemed to convey the intended meaning, so the model clearly functions.

Now, I shall take a closer look and break the whole process down into the individual steps that happen in more detail.
First, each sentence gets paired with the corresponding real translation. The tokens [start] and [end] enclose the spanish sentence - the target sequence.
This is how the text pairs look when printing some examples:

```
("Do you think I'm overweight?", '[start] ¿Crees que tengo sobrepeso? [end]')
('Turn that off.', '[start] Apágalo. [end]')
('Is the road all right to drive on?', '[start] ¿Está bien el camino para ir e
('I had an interesting day today.', '[start] Hoy he tenido un día interesante.
```

The text pairs are what the dataset is composed of and thus are split into training, validation and test data, as is usual in machine learning.

A key step for natural language processing is the vectorisation - turning the words into numerical representations that a computer can understand and work with.
In this example, for simplicity, sentences are converted to lowercase and special characters are stripped. Then, each sentence from the sentence pairs is vectorized, where individual words are replaced by tokens, or an index/reference to each word represented by a number.
This is how a sentence looks after it has been vectorized:

```
tf.Tensor(
[   6    8 2266    7  554    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0], shape=(20,), dtype=int64)
```

The encoder layer will receive an input sequence (from the english sentences), encode it, and forward the encoded sequence to a decoder layer.
The decoder additionally receives part of the target sequence (the real Spanish translation) as well and is tasked with predicting the next word in the sequence.
In essence, the transformer is supposed to learn to accurately predict the next word in a sentence of the target language (Spanish), based on the words it can see so far and the full sentence in the original language. (English)

One more piece of information that is required in order to do that, is the order and position of the words, which is done through positional embedding, where numbers tell the position of each word in the sentence, in contrast to a bag-of-words model, where the structure of the text gets lost.
There are problems when trying to use more obvious approaches, which are solved by using sinusoidal functions to create the positional embeddings. I have skimmed a great article about this topic: https://kazemnejad.com/blog/transformer_architecture_positional_encoding/
But I feel that for now, the specific reasons and maths are out of scope for my analysis and I am satisfied with understanding that this method fulfils its purpose of providing the positional embeddings to the model while avoiding certain issues.

Additionally, something that allows transformer networks to perform so well with NLP are the attention layers. These attention layers identify words and phrases that are connected and relevant to each other within a sequence and assign the strength or importance to them in the form of “weights”.
This establishes something akin to the context of the sentence, i.e. what the sentence is about and how its parts are connected, making the model analyse it a bit more like humans do.
To understand the different variations of attention layers (self-attention, multi-head attention and encoder-decoder or cross-attention) present in typical transformer networks, I researched a bit deeper and read through this article which gave me a good idea of what happens: https://towardsdatascience.com/attention-and-transformer-models-fe667f958378

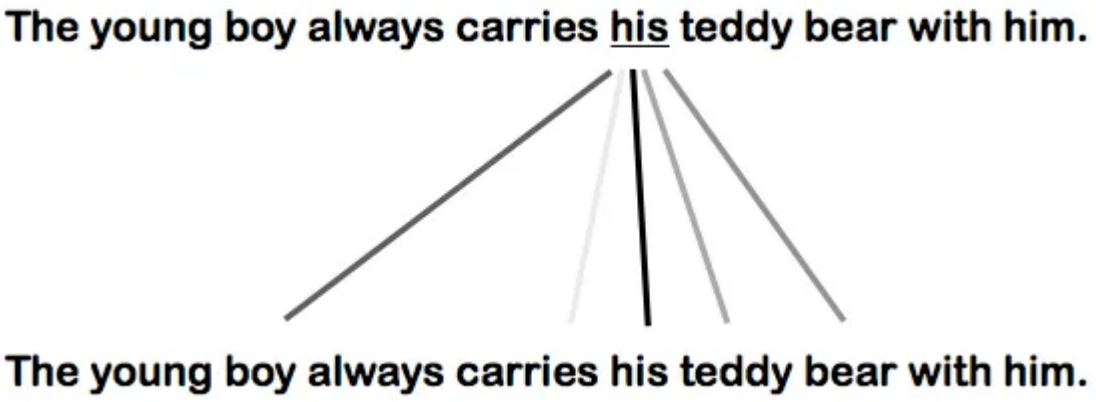
Self-attention is a form of attention just focused on a single input sequence. For that, an attention vector “z” is calculated for every word via queries, keys and values. Each word’s attention vector gives an idea about how that word relates to every other word in the sequence. So for a sequence with length n, there are not only n attention vectors, but also n attention scores calculated for each of them.

The queries, keys and values are vectors derived from their own set of respective weights multiplied by the word input x.
An attention vector z is calculated as following:

$$\text{Attention}(Q, K, V) = \text{softmax}(((Q * K^T) / \sqrt{d_k})) * V$$

Note that the query Q is constant (for the current word in the sequence), while the key K (and the value V) are iterated over, based on which word is being matched with the current word. Continuing from there, each of the intermediate attention scores are divided by the square root of the dimension of the key vector and put through the softmax function. They are then multiplied with the value vector of the word being matched and eventually summed up to get the attention vector.
This results in vectors for each word containing information about how “connected” it is to all the other words in the sequence, not unlike the weights of neurons in a neural network.

In the image below, one can see what the self-attention vector of a single word - in this example “his”, reveals. It implies how strongly the word relates semantically to other words, or how much “attention” should be directed to other words in order to understand its context.

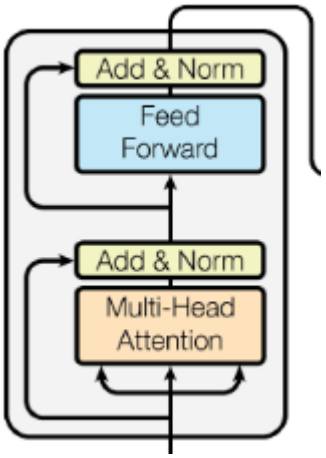


These self-attention vectors are created for every single word and they are modified throughout the training process, just as the weight matrices to calculate the q, k, v vectors are.

If that wasn’t complex enough, when speaking about attention, there is usually the so-called “multi-head attention”. The steps explained above form one “head”, so multi-head means that this process is done multiple times with different weights and then combined into one final vector (which is still multiplied with yet another set of weights).
Simply put, instead of just asking “one head’s opinion”, there are multiple attempts to establish and understand the context in different ways, aiming to extract more context information.
This is like asking multiple people to explain the meaning of a sentence and combining all their knowledge - after all someone might catch onto some important element that the others missed. So in a way, it’s a strategy to maximise the context information gained about the sentence.
The difference between encoder and decoder self-attention is the sequence that is processed (inputs vs. outputs), but in addition to that the decoder self-attention is masked, which in summary is to ensure that the model learns to predict the next word as well as possible, even without knowing the subsequent part of the sequence.
For encoder-decoder attention, information from both of the previous attention layers is merged - queries from the decoder (masked) self-attention and keys, values from the encoder self-attention.
The main distinction between self-attention and attention, is that self-attention is only concerned with attention for a single input sequence, whereas attention relates the information between the input and the target, thus combining encoder and decoder self-attention.

Layer normalisation is a form of scaling the vectors which works just like scaling data as commonly done in deep learning - by scaling the values to a certain range based on minimum and maximum values, the model can converge better because the values are confined to a common range. The approach is a bit different due to the nature of the vector embeddings, but it essentially works by adding together input and output vectors of the layer in question and then normalising.

That is what is shown in the image to the right by the connections to the layer normalisation step (yellow boxes), one is the input vector (skipping the layer in between), the other is the output vector of the layer, which are added and normalised.
Also visible are the three arrows pointing to the Multi-Head Attention, standing for the queries, keys and values obtained from the input.



For the final major layer type, there are the feed-forward layers, which are practically typical MLP-type neural networks, with an input layer, at least one hidden layer and an output layer. They take the output from the attention layers (after normalisation) and are where the more classic deep-learning style of machine learning is carried out and generally include dropout layers too. A lot of the other steps like normalisation and self-attention, multi-head attention and so on, are said everywhere to allow the model to “learn complex patterns in the data”, well as I see it, this is the part where the model actually *does* learn the complex patterns. Meaning that the other layers are mostly facilitating or enabling this learning process that is done in the feed forward layers.

For hyperparameters, the sequence length, as brought up before, is the length of an input sequence, the length is determined by the number of tokens, which can represent words, characters, or other things. For word-based tokens, a 20 word sentence would have a sequence length of 20.
Vocabulary size is the total number of words or tokens that can be accounted for by the transformer, so if the vocabulary size is too small, some words won’t be included in the vocabulary.
Batch size determines the number of input/target sequences processed, during each epoch/training step.

The embedding dimension and latent dimension hyperparameters determine the size of the respective vectors. Embedding dimension for the input sequences (typically between 100 to 1000) and the latent dimension for “hidden representations”, which are vectors created within the transformer network, e.g. the attention vectors, outputs of the feed-forward layers etc. These hidden representations are basically where the neural network tries to figure out what the inputs mean.

So, going through the whole process sequentially from start to finish, while simplifying a bit, a transformer network functions like this:
There are encoder layers and decoder layers, the number of which can vary.
First, the input sequence is converted to a numerical form (the input embeddings) and the positional information of the tokens is incorporated into these embeddings via sinusoidal functions.
Next, they are fed into an encoder layer, where the self-attention layer creates attention vectors for each token and sums them together, extracting important information about context and relationships between words (tokens). A normalisation layer scales the output vector from the self-attention layer for better model convergence. This is done after every layer in the encoder and decoder layers, so I will omit it in the following steps.
The normalised vector then goes into a feed-forward layer. The feed-forward layer is where a multi-layer perceptron fits to the data to learn and understand its features.
The decoder layer starts similarly by getting output embeddings with positional information encoded into them. The multi-head attention layer is masked this time around to stop the model from cheating by hiding data further along in the output sequence and force it to become better at predicting words based on just the known part of the output sequence.
After the attention vector from the masked self-attention has been created, some of the encoder output (keys and values) is incorporated alongside some of the decoder output (queries). These vectors are used in a final decoder-encoder attention layer to combine as much valuable

information about what was learned from the inputs and the masked outputs.
Finally, the remaining section of the decoder layer contains another feed-forward layer, fitting to the output from the decoder-encoder attention layer.
After the decoder layers, the data processed up to that point is used to generate a probability for every token from the vocabulary by applying a linear layer and finally a softmax layer.

I think what makes transformer networks special is, first of all, that they clearly work well - after all there could be tons of potential structures for neural networks, but they might not work well in practice.
But apart from that, they also have several unique advantages.
First, they do a really good job at maximising the contextual information gained from sentences in the attention layers. On top of that, they are designed cleverly in a way that allows parallel processing in multiple ways: For the multiple heads in the attention layers, for both the encoder and decoder layers and finally for the feed-forward layers.
The combination of masked self-attention for the outputs and self-attention for inputs also appears to be highly effective at training the transformer network to function well.

All in all I think that this exercise has shown me that transformer networks are well-designed and not quite as difficult to understand as it might seem from the well-known illustration about its architecture. The low-level maths behind it can be a bit intimidating, but I actually find that the positional encoding seemed more difficult to grasp than the attention layers, which ultimately just creates vectors with many dimensions due to the amount of information that is extracted in the process

Tom Cordruwisch
15.05.2024
R504D80 Deep Learning

I think I can also now say that transformer networks, as is often said about them, are indeed kind of like a “fancy autocomplete”, albeit a very fancy one with much better outputs, in large part due to the attention layers. But in the end, no matter how complex and sophisticated the structure is, a transformer does choose the most likely contender for what should follow a certain sequence of words or characters. (Technically it gives probabilities for all the tokens, but effectively the most likely token is what matters.)
But, I would argue that humans are not really all that different, after all we also construct sentences from smaller pieces and use our contextual understanding and intuition to determine the next piece.
What separates us from an AI for now, is that we usually have an intention, a personality and life experience, which transformers lack.