ABSTRACT

In 2017, Google published a paper by the title “Attention Is All You Need “ which they claimed to be an alternative to Recurrent Neural Networks and Convolutional Neural Networks in Natural Language Processing tasks. That architecture which they named *Transformer* took the world by storm and now architectures using it, like BERT by Google, GPTs by OpenAI, are known to give state-of-the-art results in NLP tasks. My efforts have focused on constructing the Transformer from scratch using Pytorch and to some extent, Numpy both of which are scientific computing libraries in python.

In this report I have followed a bottom-up approach. A bottom-up approach is where one starts with the elementary parts, delineating their roles and then brings them together to form higher level compound modules. The process is then repeated one level after the other in a hierarchical manner till we are left with only one module, that is in this case, the Transformer.

A point to note is that Pytorch, which is a machine learning library, comes with its own Transformer module, but none of the high level APIs have been used here and everything has been built using pytorch tensors (vectors, matrices, etc) and basic operations that can be performed on them like matrix-multiplication, scalar product, etc.

As mentioned earlier, the transformer can be used for many of the NLP tasks.

Some details might vary depending on the task. The task that my efforts have dealt with here is Language Translation. All the work done is available publicly at https://github.com/shouvikcirca/Transformer .

INTRODUCTION

Prior to the publishing of this paper, most deep learning models that gave state-of-the-art results in NLP tasks used variations of either Recurrent Neural Networks (RNNs) or Convolutional Neural Networks (CNNs). Although both these architectures have proved to be tremendously useful and accurate, they do not come without shortcomings.

CNNs are easily parallelizable but cannot capture variable length sequential dependency very well. When we translate a sentence from one language to another the target sentence need not be equal in length to the source sentence and the dependency of the next word on the previous words in terms of degree of emphasis to be given to each of the preceding words, the number of which may vary, is not easy to handle using CNNs.

RNNs do the job of handling long range dependencies very well, particularly variants like Long-Short-Term-Memory networks and Gated-Recurrent-Units. However there is an inherent sequentiality in their very architecture. RNNs strive to figure out the sequentiality or the temporal aspect in the input data they work on.(Language has a temporal aspect because tokens follow or precede other tokens). They incorporate a feedback mechanism in which the outputs of a time-step also serve as inputs to the next time step. This makes them non-parallelizable.

To combine the best of both worlds, folks at Google came up with an architecture they called *Transformer* which handles token dependencies using a mechanism called *attention* and they take away the sequential aspect from the architecture itself and use a trick to encode sequentiality in the data itself in a way that they don't lose out on parallelizability. Hence, the working of a transformer can be parallelized.

PREPROCESSING

The data here comprises sentences in English and German. The first step therefore would be to convert them to numeric representations because that is what Machine Learning models crunch on, numbers.

Our dataset consists of 2 subsets. One set E contains n english sentences. The other subset G contains their corresponding m German translations, where m = n. We tokenize each sentence which means that each sentence is converted to a collection of entities (which I'll call a token-list) which could be words, punctuation marks and so on depending on the tokenizer used. For example, if we had a sentence " I have a cat ", it is now converted to the token-list ['i', 'have', 'a', 'cat']. I use the tokenizer that comes with the NLTK library in Python. For simplicity I also convert all tokens to lowercase.

Each token-list will have a few appendages.

1) An *eos* token token at the end of every sentence to mark their ends
2) An *sos* token at the beginning of every German sentence. This is needed for inference and will be explained later.
3) The source sentence and its target translation, taking into account the sos and eos tokens, have to be of equal length else the process will break in one of the attention layers in the decoder due to dimensional inconsistency. That is why *pad* tokens have to be added to either of source and german sentences after the eos tokens. Though the transformer is meant to handle translations of variable lengths, we put a limit on the length over which the translation can span at the most.

   What I have done is made every token-list have a length of 30 (All of the original sentences had fewer than 30 tokens) by adding pad tokens. Though this approach is a bit inefficient, I went for it because it is easier to do and so that I can dedicate more time on constructing the transformer.

That gives us n+m token-lists each having dimensionality 30X1.

EMBEDDINGS

The earlier approach to numerically representing tokens was *one-hot encoding*. Assume for a moment that we had just 3 sentences in our dataset

a. The dog ran
b. The cat jumped
c. The dog barked

The one-hot approach involved constructing a template vector like the one shown below, each index representing a token within the entire dataset. Each index is a feature or dimension.

the	dog	cat	jumped	ran	barked

The numeric representations for the sentences would now be

a. 110010
b. 101100
c. 110001

If we were doing a translation from some other language to English and the model had generated two words ‘the’, ‘cat’, there is nothing to tell the model that ‘ran’ is a better fit than ‘barked’ for the next to-be-generated token since b is equidistant to both a and c if we were to take the euclidean distance as the distance metric.
The solution then would be to ‘embed’ the words in a e-dimensional space where ‘jumped’ is closer to ‘ran’ than it is to ‘barked’. Now each of the tokens is represented using a e-dimensional vector. And the elements now instead of being 1s or 0s, are floating point numbers. These vectors specify the positions of the corresponding words in the e-dimensional space as illustrated below. In the figure, we take a 2-dimensional space for convenience of visualization. But in the paper 512-dimensional vectors have been used. In my implementation I have used 10-dimensional vecs. These values are not handcrafted but rather learned using machine learning models like Skip-Gram and Continuous Bag Of Words. I have used the latter model made available by the Gensim library for creating vectors for the tokens.
The token-lists are thus converted to vectors which I’ll call embeddings. So now [‘i’, ‘have’, ’a’, ’cat’] looks something like

```
[       [-0.3094,  0.1370, -0.8961, -0.5595,  0.3625,  1.2762, -0.1480, -0.0423, 1.4488, -2.3458],
        [ 0.4460, -1.7436,  0.8294, -0.6769,  0.5262, -1.5352, -0.0522, -0.1299, 0.3601,  0.0895],
        [-1.2254, -1.7075,  2.1326,  2.3426, -1.4736, -1.0597,  1.3589, -1.1039, 1.8930, -1.3084],
        [-1.1226, -0.6637,  0.9045,  1.4180,  0.8138,  0.8162,  0.2994,  0.1133,  -0.5834, -0.7878]       ]
```

FIGURE OF EMBEDDING SPACE

On converting the token-lists to embeddings, we now have n+m embeddings each having a dimensionality 30X10.

POSITIONAL ENCODING

I mentioned earlier about encoding sequentiality into the data itself. We do that using positional encoding. All the embeddings have to go through this process. Below are the equations for doing it.

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

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

Over here, dmodel is the embedding dimensionality, i.e, 10. For every even index position in an embedding we replace the value with (1) and for every odd position we use (2). pos is the position of the corresponding word in the corresponding token-list. If we consider the example from earlier**,** the vector for 'a' is
[-1.2254, -1.7075, 2.1326, 2.3426, -1.4736, -1.0597, 1.3589, -1.1039, 1.8930, -1.3084]
The element at index 3 will be

$$PE(2,3) = \cos\left(2/10000^{3/10}\right)$$

The position of 'a' in the token-list is 2. This process gives the positional encodings. The authors haven't given a specific justification for using this mode of encoding and have explicitly stated that other approaches could also be tried.
We take the positional encodings and elementwise-add them with their embeddings which gives us positionally encoded embeddings. I will refer to a positionally encoded embedding as **p2e** henceforth.
This leaves us with (n+m) p2es.

ATTENTION
The job of this mechanism is to find out how much each of the tokens in a given sentence s2 is dependent on each of the tokens in a given sentence s1. s1 may or may not be the same as s2.
The core component in the attention mechanism is the attention layer. An input to the attention layer is called *query*. The queries here would be the p2es of a given sentence. The very first attention block in the encoder is a self-attention block, meaning s1 = s2.

For our example sentence we have 4 queries. let's call them q1, q2, q3 and q4. Each token in a sentence will also have 2 more vectors assigned to it which are called the *keys* and *values* respectively. That gives us k1, k2, …, k4 and v1, v2, ……, v4.
If we take q1 and find out scaled dot products with respect to all the keys and apply the softmax function over them, that gives us the degree of attention the token in s2 corresponding to q1 is paying on all the other tokens in the sentence s1 since softmax squashes its input to a probability distribution. The operations stated above would give us four scalar numbers which I'll call **qks**. These qks, namely, qk1, qk2, ….qk4 are associated with the tokens and their summation yields 1.0 as result. The diagram below should make this picture more clear. These qks could be interpreted as the weightages given to the values. We multiply each qk with its corresponding value vector and add them all.

DIAGRAM FOR ATTENTION

This converts each p2e to a new vector, an encoder self-attention vector, which I'll refer to as an **esa** vector. In context of our example sentence, we have now four esa vectors esa1, esa2, …., esa4. The transformation of the p2es to esas are independent of each other and thus can be parallelized. The generation of each of the esas can be done together in parallel.

The queries that go into the self-attention block do not have the same dimensionality as the p2es. The p2es are projected to a lower dimensionality using linear layers and thus the dimensionalities of the queries, keys and values are reduced. This is done for computational reasons which I'll be getting into in a while. In the paper 512-dimensional vectors are made 64-dimensional. I have reduced the dimensionality from 10 to 5. This is done by multiplying each p2e of size 1X10 with a weight matrix W^Q with dimensions 10x5 to get the query, a weight matrix W^k with dimensions 10x5 to get the key and a weight matrix W^v with dimensions 10x5 to get the value. The apt values for these weight matrices are learnt during the learning process.

The transformation process from p2es to esas is encapsulated within what the authors have called an *attention head*. We use multiple attention heads. The authors have used eight attention heads because they reduced the size of p2es by a factor of 8 before feeding the queries to the attention block. Since I have reduced the dimensionality by a factor of 2, I have used two attention heads which again can work parallelly. This gives us, for every p2e, two esas of reduced dimensionality. We concatenate them to get back a vector of same dimensionality as the p2e. I'll refer to this synthesized esa as a **sesa**. We have now sesa1, sesa2, …., sesa4. Each sesa of size 1X10 is multiplied with a weight matrix W^O to get a projected sesa. These weights also have to be learned.

LAYER NORMALIZATION AND SKIP CONNECTIONS

Each projected sesa is elementwise-added with its corresponding p2e. This is called a *skip connection* where the input to a cascade of layers is added with the output emerging from the end of the cascade. The resulting vector is then layer-normalized. *Layer Normalization* is a technique that normalizes the data input at each layer by taking the mean over the activations in a particular layer. Both these techniques pertain to the field of Deep Learning. Skip connections help build deeper networks without *degradation* in performance and the other one makes convergence faster. The output from here still contains 1X10 sized vectors which are then fed to the feedforward block.

FEEDFORWARD BLOCK

This block contains a Feedforward Neural Network with a few layers followed by ReLU activations. The dimensionality of the vectors is retained in the output of this network. The

output is elementwise-added to the input and the resultant vector is layer normalized. I'll call the vectors obtained encoding layer vectors or elas.

ENCODER

The transformation from p2es to elas can be encapsulated within what is called an encoder block. We can use multiple encoder blocks. The authors have used six. The output of one encoder block, i.e an ela, serves as input (like the p2es) for the next encoder block. The output of the last encoder block is used by all the Decoder blocks.

DECODER

The inputs, just as in the Encoder, are p2es of the german sentence tokens. These p2es are fed into an attention block. The attention block performs the same fundamental operation but is more constrained than the one in the encoder. Since German is the target language here, a word cannot pay attention to the words that follow it because conceptually their generation is dependent on the current word. Hence a token in the target sentence can only attend to words preceding it, i.e, words that have been generated before it. Thus we need a way to distribute the probability only among the words leading to the current word, itself including, so that the weightages placed on the words following the current word become 0.

MASKED ATTENTION

One thing that needs to be assured for the proper working of the method I am about to describe is that the values in the p2es are positive. That could be ensured by applying the mod function on the p2es. But one has to be careful because the mod function is not differentiable at x=0.

Also the weights in W^Q, W^K and W^V need to be all positive. Not doing this will result in undefined values.

If we consider the 3rd token, the query for that would be q2 but the keys that it will operate on are as follows.

k1, k2 and k3 will remain as they are but k4 and k5 will each become $[-\infty, -\infty, -\infty, -\infty, -]$. The qks we get then are five numbers such that qk1, qk2 and qk3 are real positive numbers but qk4 and qk5 hold the value $-\infty$. When we apply a softmax to these value qk4 and qk5 are changed to 0 because $e^{-\infty}$ approximates to 0. We are left with a probability distribution over q1, q2 and q3 summing to 1. Henceforth the values for the 4th and 5th token are given no weightage.

I'll call the vectors we get from this decoder self-attention vectors or **dsas.** The same procedure involving multiple heads is used .The synthesized dsas are elementwise-added to the p2es and layer normalized. The output is then fed to the next attention block.

NOT SELF-ATTENTION

The two attention blocks described before this are called self-attention blocks because as mentioned earlier, s1=s2. However that is not the case in this one. This block tries to determine how much each token in the target sentence should pay attention to the tokens in the source sentence. In other words, s2 = target sentence(German) and s1 = source sentence (English). It does not have the restrictions imposed in the masked self-attention block. However there is a difference in the vectors it operates on. This is where the output of the last Encoder layer comes into play. The queries are formed by projecting the input from the previous layer in the decoder but the keys and values are formed by projecting the outputs of the last encoder layer.

The synthesized vectors from here are further projected as in the Encoder Attention block. They are then elementwise-added to the input to this attention block and layer normalized. It is for this layer that the size of the source sentence embedding and that of the target sentence embedding must be the same. If that is not taken care of, the architecture breaks here. The output is fed to a feed forward network that retains dimensionality and again elementwise-added to the input to the feedforward network and layer normalized.

The transformation uptill here starting from the p2es of the target sentence is encapsulated within a decoder block. The paper makes use of six decoder blocks, feeding the output of one to the next.

PROJECTION TO VOCABULARY

The output of the final decoder layer is passed through a linear layer. (A feedforward network is composed of multiple linear layers). The linear layer projects the output from the final decoder layer to a one-hot vector where each index represents an entity from the german vocabulary. The vocabulary is a set comprising all entities that can be isolated in the given corpus, in this case, all the german sentences used. For example, for a german vocabulary consisting of 550 entities and an input of dimensionality 30X10, the output will be 30X550. The output is a probability distribution over the words in the vocabulary.

TRAINING

We also need a target vector to enable the learning process. I take the German sentence that goes into the decoder after being converted to p2es and convert into a one-hot vector. I'll call this the ground truth vector. Using this and the prediction vector, we can calculate a loss function like Cross-Entropy loss, KL Divergence loss, mean squared error, etc. For a given sentence pair, we'll get thirty loss values. We could take their sum or the mean by dividing the sum by 30. This loss value is then backpropagated and all the weights in the weight matrices that we have used so far get updated accordingly as the training continues.

INFERENCE

Inference refers to the process of using a trained model to make the prediction. This portion of the work is conjectural on my part owing to non-success in isolating a source that provides a proper explanation.

The entire english p2e goes into the encoder. But the p2e that goes into the decoder corresponds to a token-list having 30 sos tokens. The output that is given out corresponds to a german sentence having 30 tokens. We take the first token t from this and replace the first sos token in the german input p2es with the p2e for t1 and go for the next run of the transformer. This time we take the second token t2 from the output and replace the second p2e with the p2e for t2. Now the token-list corresponding to the input p2es would be [t1, t2, sos, sos]. This is repeated till we have either generated tokens upto t30 or we get an eos token as output. The diagram below should help in understanding the process better.

DIAGRAM FOR INFERENCE

BLEU SCORE

The BLEU (Bilingual Evaluation Understudy) score is a metric to evaluate how well a model performs. For this we need reference translations. The BLEU score is a real number that lies in [0,1]. For it we have a reference sentence *ref* and the obtained translation *trans* . The perks of this metric are that it is not computationally intensive and that it is easy to calculate. I have also made a code snippet for calculating the BLEU score and is available at https://github.com/shouvikcirca/Transformer .

CONCLUSION

We are done finally and you have the Transformer. It is a very heavy model and takes a lot of resources in terms of hardware, money and time to train if we are willing to get a decent performance. The objective of this undertaken effort, i.e, constructing the transformer from scratch, has been achieved. This architecture was introduced nearly three years ago and a lot of developments have taken place since. If one wants to use Transformer models, there are some fantastic libraries out there that are in synchronization with the developments in the literature and

that are also highly optimized like Google BERT, OpenAI GPT, Pytorch Transformers, etc. It has been a great learning experience for me.