Understanding Transformers — Encoder

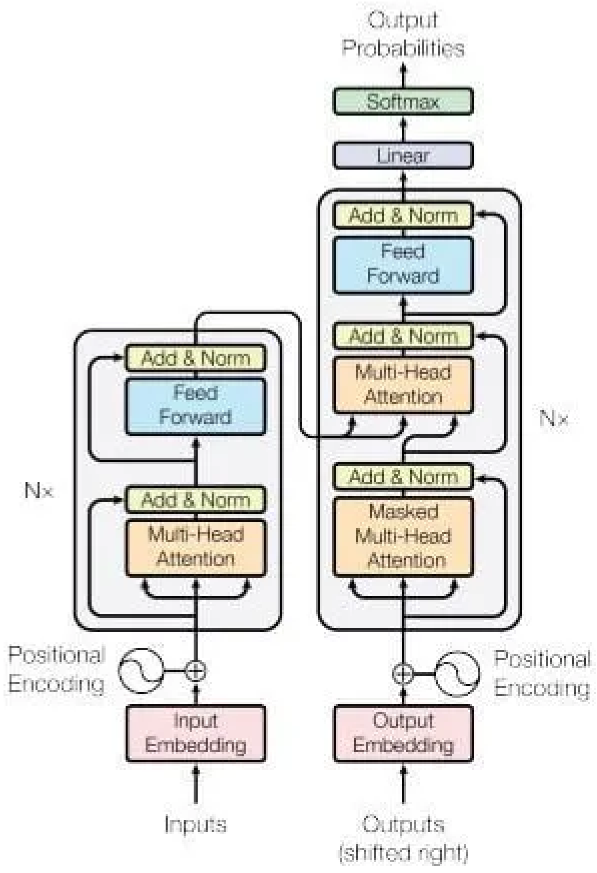
Sathya Krishnan Suresh · Follow
Published in MLearning.ai · 7 min read · Oct 12, 2022

Sathya Krishnan Suresh, Shunmugapriya P

Transformers were introduced in the year 2017 and since then, there has been an explosion in the number of the advanced models-relating to transformer architecture-developed and the uses of these models have really increased the importance of Natural Language Processing even more. Transformers have been able to solve most of the common tasks of NLP with the highest efficiency and the researchers are finding more and more problems where Transformers can be applied. With this introduction, let's talk about the transformer architecture and mainly the encoder part of the architecture. The code for the article is given here.

Transformer Architecture:

A transformer is made up of an encoder and a decoder and were originally designed for sequence to sequence tasks like machine translation, question answering, etc. The architecture of the original transformer model is given here.



Transformer architecture

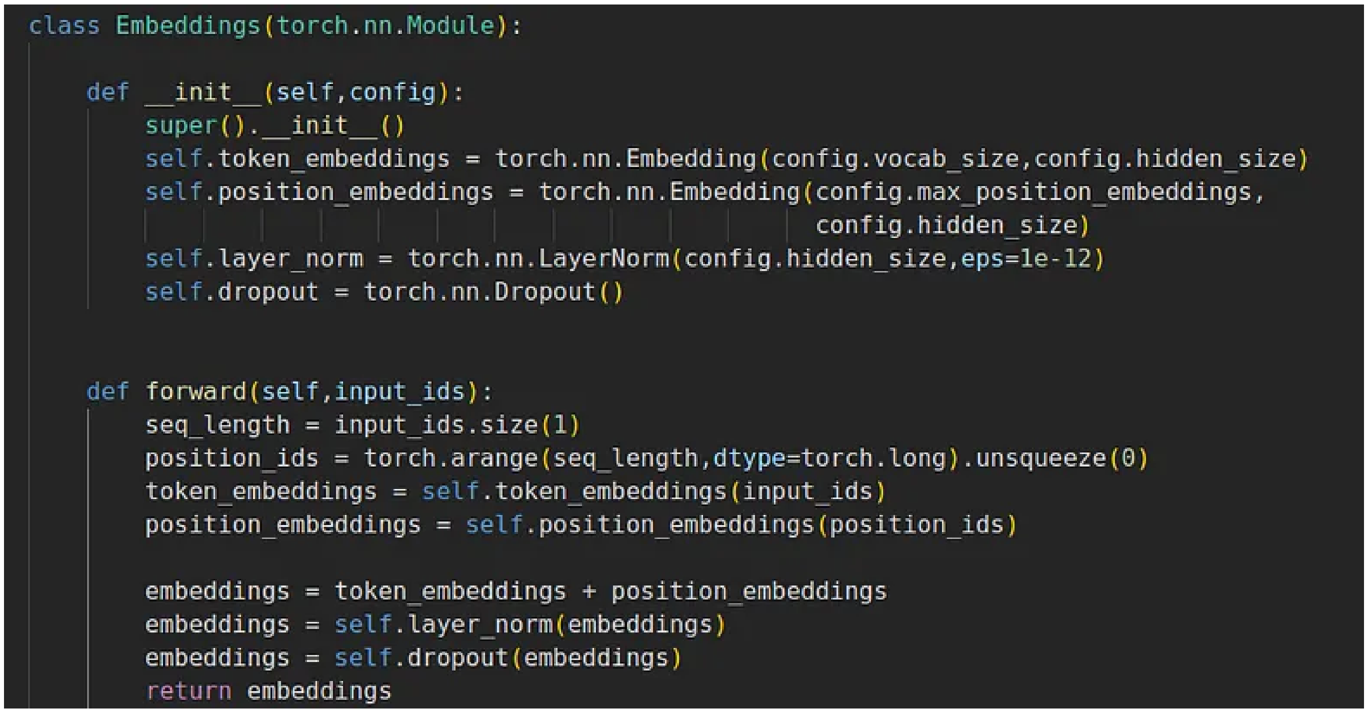
The entire grey rounded box on the left is the encoder and similarly on the right is the decoder. The encoder takes in the embeddings which is the sum of the regular embeddings and positional embeddings and outputs tensors of the same shape but those tensors have a lot of information-contextual meaning, part of speech, position of the word, etc-encoded in them. The tensors output by the encoder are often called the *hidden state*. These tensors are fed to the decoder and they decode the hidden state depending on the task it has to solve and it’s pre-training objective.

The architecture might look complicated but let's break it down layer by layer and build the architecture from bottom up.

Embeddings:

The raw text data that we have cannot be used to train transformer models as they understand only numbers. Standard operations like tokenization and one-hot encoding were used earlier but none of them were able to fully encode the information from the text into the numbers. Embeddings were the first one to encode understandable information into those numbers or the correct way of saying would be vectors. So rather than passing the tokens to the encoder, embeddings of it are passed. But one major problem was encoding the information about the position of the words in the embeddings. Standard embedding just mapped each token to an n-dimensional vector but it never encoded the position information.

feeding the tokens of the words as inputs we feed the position ids or indices of each sentence as inputs. This enables the position embedding layer to give an useful representation of the positions. The final output of the embeddings layer is generated by adding the token embeddings and position embeddings and applying a normalization to the result so that there are no huge values. The code for the embeddings layer is given below.



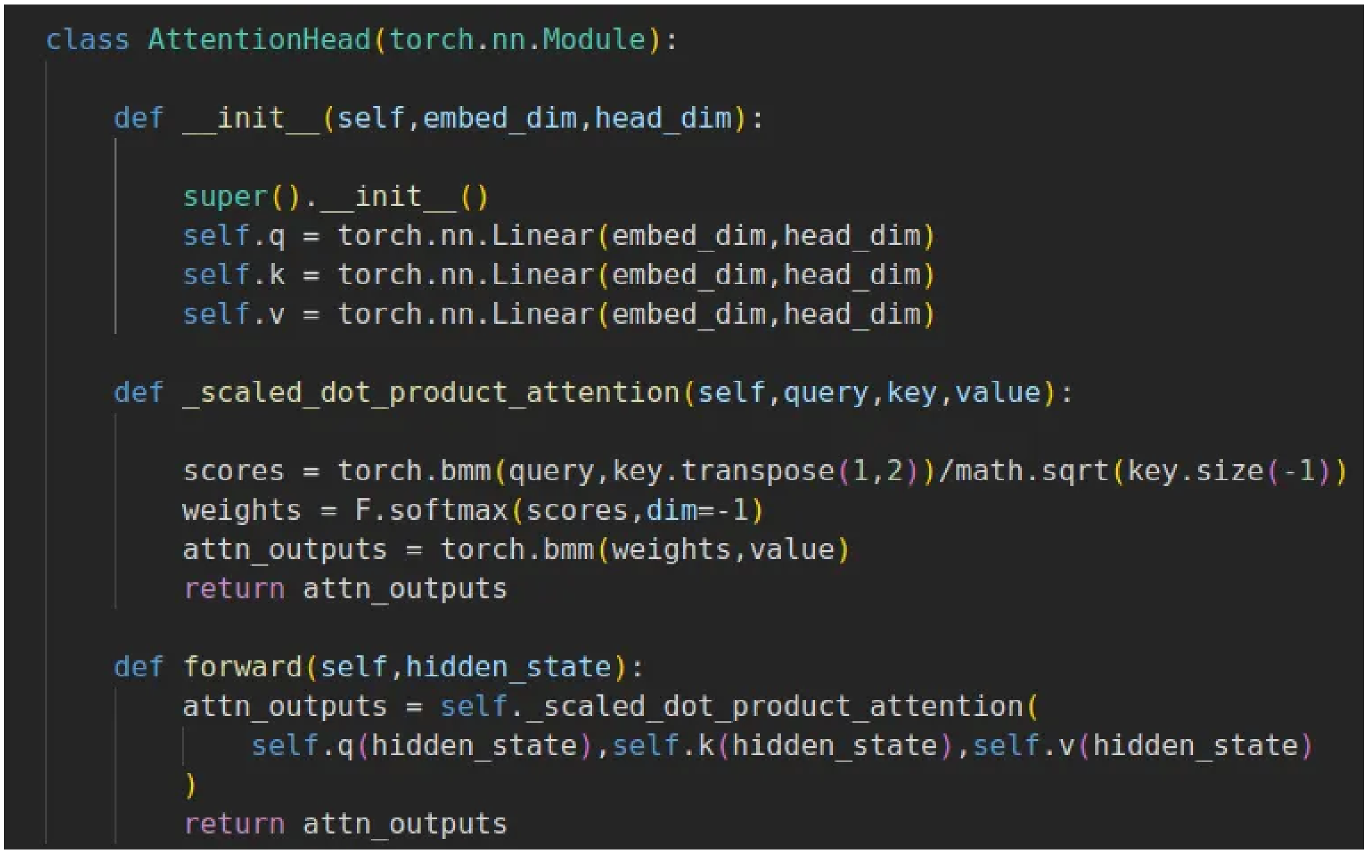
```python
class Embeddings(torch.nn.Module):

    def __init__(self,config):
        super().__init__()
        self.token_embeddings = torch.nn.Embedding(config.vocab_size,config.hidden_size)
        self.position_embeddings = torch.nn.Embedding(config.max_position_embeddings,
                                                      config.hidden_size)
        self.layer_norm = torch.nn.LayerNorm(config.hidden_size,eps=1e-12)
        self.dropout = torch.nn.Dropout()


    def forward(self,input_ids):
        seq_length = input_ids.size(1)
        position_ids = torch.arange(seq_length,dtype=torch.long).unsqueeze(0)
        token_embeddings = self.token_embeddings(input_ids)
        position_embeddings = self.position_embeddings(position_ids)

        embeddings = token_embeddings + position_embeddings
        embeddings = self.layer_norm(embeddings)
        embeddings = self.dropout(embeddings)
        return embeddings
```

Self-Attention:
The embeddings that are fed to the encoder are static and have some amount of information encoded in them. But the self-attention layer takes into account, the context of the words or tokens, in a sentence or a set of sentences and encodes those information too in their respective embeddings.

Encoding the context into the embedding is an important task because homonyms in a passage will be mapped to the same embeddings. For example, consider a passage that contains the term 'bat'. 'Bat' in the passage can be used in the sense of a mammal or in the sense of a stick with which players play or it maybe used in both the senses. The embeddings fed into the encoder will have the same embedding value for both the senses but we do not want that. When the self-attention layer is done with the embeddings passed to it, the embeddings obtained will have their context encoded in them. For example, 'bat' will have more of the mammal sense instead of the stick sense if it has been used in the passage like 'Bats live in caves'.

Self-attention encodes the context by using three tensors — **Query, Key and Value** — and some mathematical operations relating to those tensors. The tensors are generated by using a learnable linear projection of the embeddings passed to the layer. The meaning for these tensors can be understood with a simple example. When you search for 'Infinity war' in Google, the search text that you are entering is called a query. Google's servers would have hash keys for similar queries and those keys are matched with the query to find the similarity scores. Those hash keys are the keys in discussion. Finally content returned to the user is the value.

In the standard attention mechanism the query comes from the decoder and key, value pair is provided by the encoder. But in self-attention mechanism the query also comes from the encoder. The matrix multiplication between the query and key tensors gives the similarity score of a token(word) with all the other tokens(words). Since the values resulting from a matrix multiplication can be large, softmax is applied to the score matrix and finally it is multiplied with the value matrix to generate the outputs of this layer. Given below is the code for the attention layer in PyTorch.

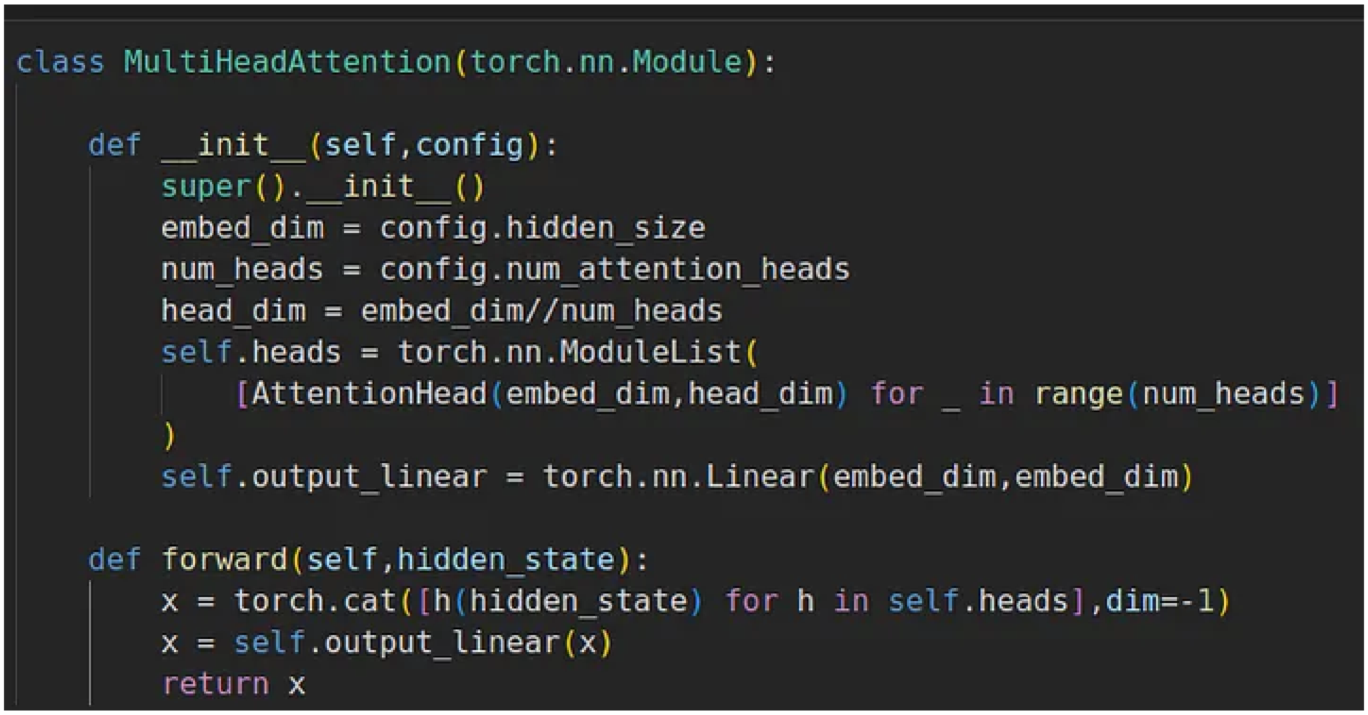


```python
class AttentionHead(torch.nn.Module):

    def __init__(self,embed_dim,head_dim):

        super().__init__()
        self.q = torch.nn.Linear(embed_dim,head_dim)
        self.k = torch.nn.Linear(embed_dim,head_dim)
        self.v = torch.nn.Linear(embed_dim,head_dim)

    def _scaled_dot_product_attention(self,query,key,value):

        scores = torch.bmm(query,key.transpose(1,2))/math.sqrt(key.size(-1))
        weights = F.softmax(scores,dim=-1)
        attn_outputs = torch.bmm(weights,value)
        return attn_outputs

    def forward(self,hidden_state):
        attn_outputs = self._scaled_dot_product_attention(
            self.q(hidden_state),self.k(hidden_state),self.v(hidden_state)
        )
        return attn_outputs
```

Self-attention code

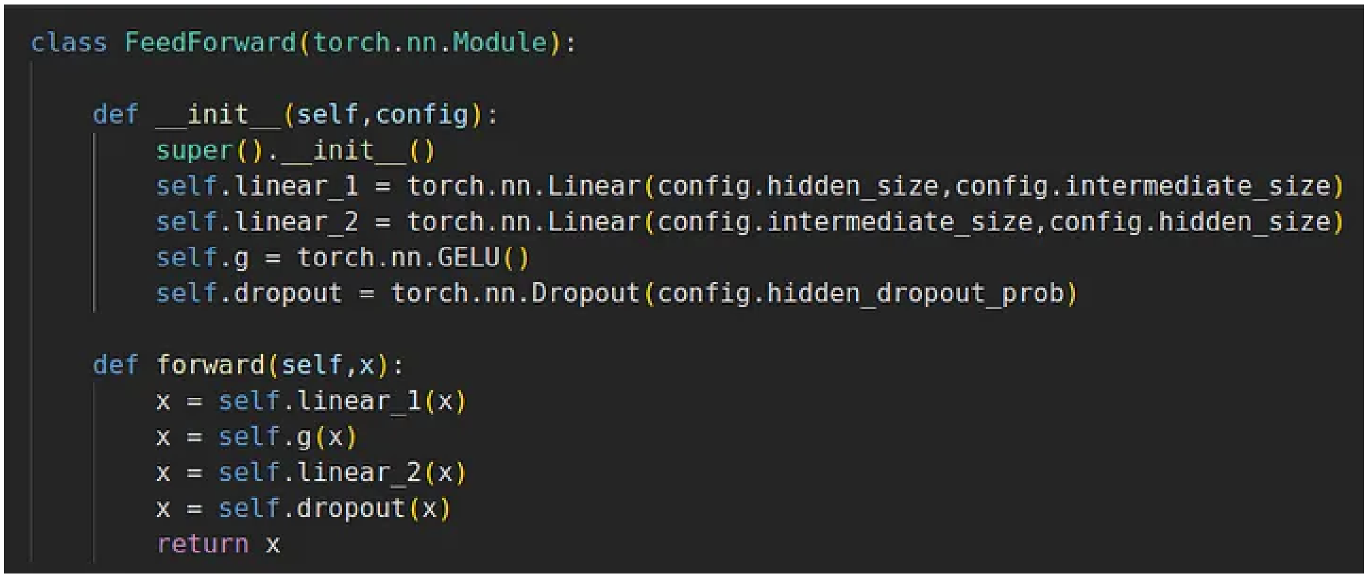
Multi-Head Attention:
Basically, Multi-Head Attention is made up of a number of self-attention units. This is because each attention head will get to focus on a particular feature of the text. One head can get to focus on the subject-verb relationship and the other might get to focus on the tense of the text and so on. This is similar to using multiple filters in a single convolutional layer and as we know from ensembling, multiple models more often than not lead to good results.

Usually in Multi-Head attention the last dimension — embedding dimension — is split equally among the attention heads for scalability and the outputs of the attention heads are concatenated and a linear transformation is applied on those outputs to get the final output. The linear transformation is applied so that the output generated will be suitable for the feed-forward layer to which it is passed later. The code for this layer is given below. The important point here is to take care of the head dimension and concatenating them back.



```
class MultiHeadAttention(torch.nn.Module):

    def __init__(self,config):
        super().__init__()
        embed_dim = config.hidden_size
        num_heads = config.num_attention_heads
        head_dim = embed_dim//num_heads
        self.heads = torch.nn.ModuleList(
            [AttentionHead(embed_dim,head_dim) for _ in range(num_heads)]
        )
        self.output_linear = torch.nn.Linear(embed_dim,embed_dim)

    def forward(self,hidden_state):
        x = torch.cat([h(hidden_state) for h in self.heads],dim=-1)
        x = self.output_linear(x)
        return x
```

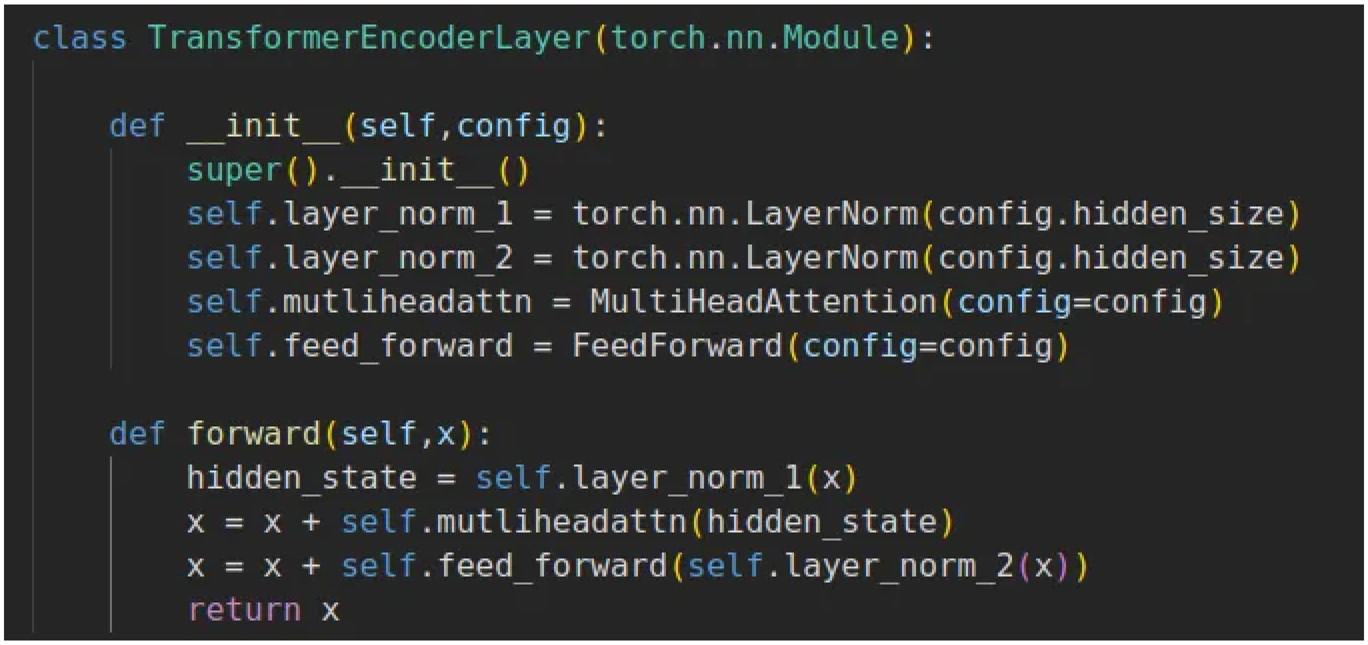
Feed-Forward Layer:
The Feed-Forward layer is a simple layer composed of two linear layers, a GELU layer and a dropout layer. In this layer the embeddings are processed independent of each other and it is thought that this is the layer where most of the memorization of the information takes place. Hence whenever transformer models are scaled up, usually the feed-forward layer is the one that is scaled up the most.



```
class FeedForward(torch.nn.Module):

    def __init__(self,config):
        super().__init__()
        self.linear_1 = torch.nn.Linear(config.hidden_size,config.intermediate_size)
        self.linear_2 = torch.nn.Linear(config.intermediate_size,config.hidden_size)
        self.g = torch.nn.GELU()
        self.dropout = torch.nn.Dropout(config.hidden_dropout_prob)

    def forward(self,x):
        x = self.linear_1(x)
        x = self.g(x)
        x = self.linear_2(x)
        x = self.dropout(x)
        return x
```

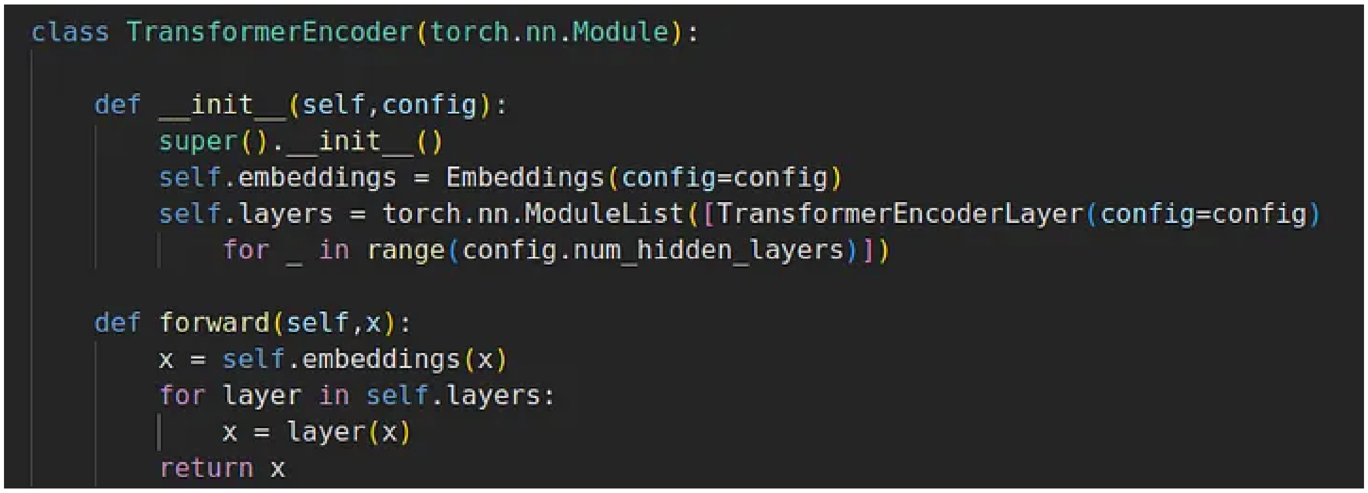
Transformer Encoder Layer:
All the layers needed for an encoder layer have been developed and it's just a matter of putting them together into a single model. But before just blindly following the architecture diagram given above, we should think about the positioning of the skip layer and the normalization layer.

Depending on the position of the normalization layer, there are two types of normalization— Post-layer normalization and Pre-layer normalization. In the former type, layer normalization is applied between two skip connections and the original architecture given above follows this type. In the latter type, the layer normalization is applied within the span of the skip connection (this will be clear when you look at the code). Most of the architectures being used now follow the latter because in the post-layer normalization the weights and gradients diverge, and subsequently training becomes very difficult. The code given below also follows the latter type of normalization.



```
class TransformerEncoderLayer(torch.nn.Module):

    def __init__(self,config):
        super().__init__()
        self.layer_norm_1 = torch.nn.LayerNorm(config.hidden_size)
        self.layer_norm_2 = torch.nn.LayerNorm(config.hidden_size)
        self.mutliheadattn = MultiHeadAttention(config=config)
        self.feed_forward = FeedForward(config=config)

    def forward(self,x):
        hidden_state = self.layer_norm_1(x)
        x = x + self.mutliheadattn(hidden_state)
        x = x + self.feed_forward(self.layer_norm_2(x))
        return x
```

Transformer Encoder:
We have everything needed to implement the encoder part of the architecture. First we generate the embeddings from the input tokens given and then pass those embeddings through a stack of the encoder layers discussed above. The code again is simple and is given below.



```
class TransformerEncoder(torch.nn.Module):

    def __init__(self,config):
        super().__init__()
        self.embeddings = Embeddings(config=config)
        self.layers = torch.nn.ModuleList([TransformerEncoderLayer(config=config)
            for _ in range(config.num_hidden_layers)])

    def forward(self,x):
        x = self.embeddings(x)
        for layer in self.layers:
            x = layer(x)
        return x
```

This encoder layer alone can be used as a stand-alone model for a lot of tasks like text classification, masked language modelling, etc by just adding a suitable task dependent body on top of the encoder block.

Conclusion:

The complete structure of the encoder layer has been discussed in this article. Decoder layer will be discussed in the next article. I hope you had fun reading the article as much as I had while writing it.

Mlearning.ai Submission Suggestions

How to become a writer on Mlearning.ai

medium.com

Machine Learning · Deep Learning · NLP · Transformers · Ml So Good

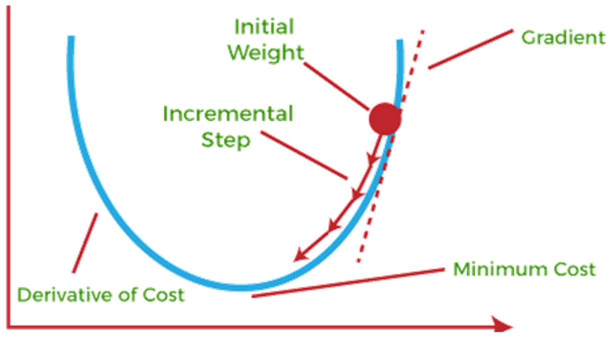
Written by Sathya Krishnan Suresh

Follow

74 Followers · Writer for MLearning.ai

Computer Science sophomore. ML, AI and NLP are my fields of interest LinkedIn: https://www.linkedin.com/in/sathya-krishnan-suresh-914763217/

More from Sathya Krishnan Suresh and MLearning.ai



Sathya Krishnan Suresh in MLearning.ai

Understanding Adaline

Contributors: Shunmugapriya, Sathya Krishnan Suresh Github link:...

6 min read · Jan 23, 2022

67

Fabio Matricardi in MLearning.ai

How I Built a Chatbot that Crushed ChatGPT with Zero Cost AI Tools

Challenge Accepted! How I created a chatbot that surpassed the performance of the...

· 9 min read · Feb 4, 2024

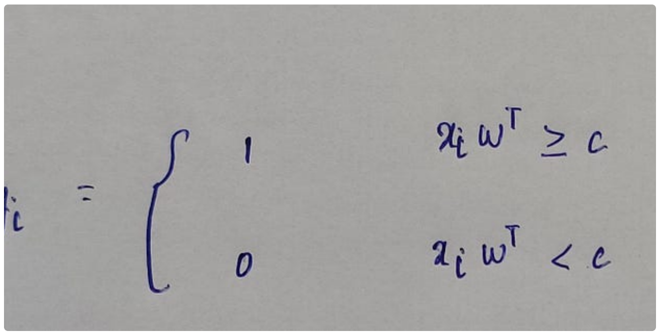
1.6K 5

Tomer Gabay in MLearning.ai

How to Setup Your Macbook for Data Science in 2024

Easy Steps to Get the Best Experience From Your MacBook as a Data Scientist

· 6 min read · Jan 18, 2024



Sathya Krishnan Suresh in MLearning.ai

Understanding Perceptron

In this article let's take a look at one of oldest classification algorithm in machine learning-...

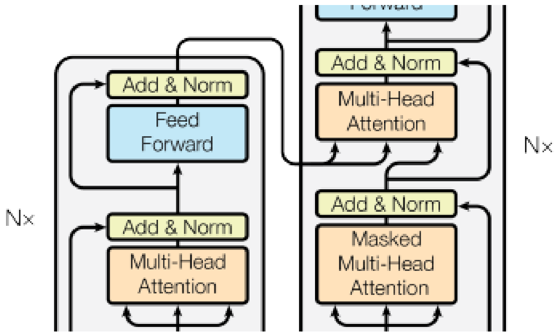
6 min read · Jan 5, 2022

565 4 58

See all from Sathya Krishnan Suresh

See all from MLearning.ai

Recommended from Medium



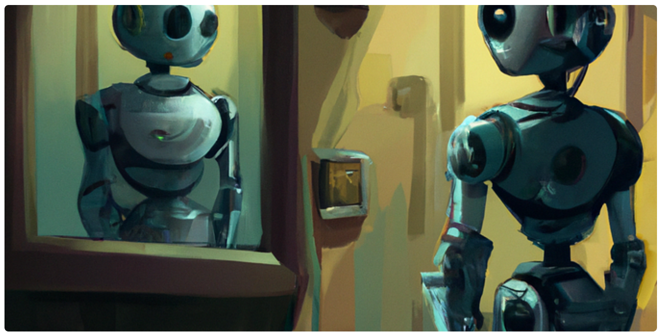
Dr. Ernesto Lee

An Intuitive Explanation of 'Attention Is All You Need': The...

This is the technology that makes ChatGPT works!

9 min read · Oct 14, 2023

28



Thomas van Dongen in Towards Data Science

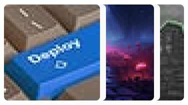
Demystifying efficient self-attention

A practical overview

20 min read · Nov 7, 2022

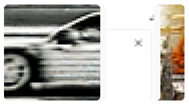
623 2

Lists



Predictive Modeling w/ Python

20 stories · 904 saves



Natural Language Processing

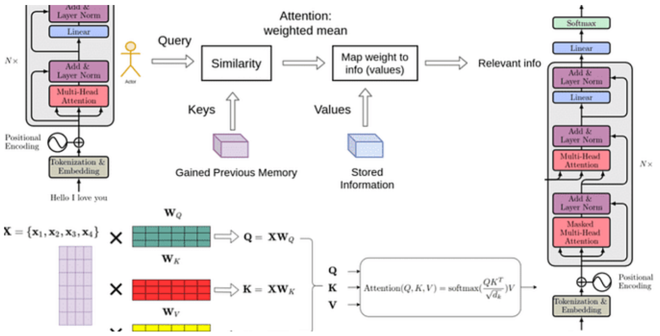
1200 stories · 670 saves

Practical Guides to Machine Learning

10 stories · 1059 saves

The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 306 saves



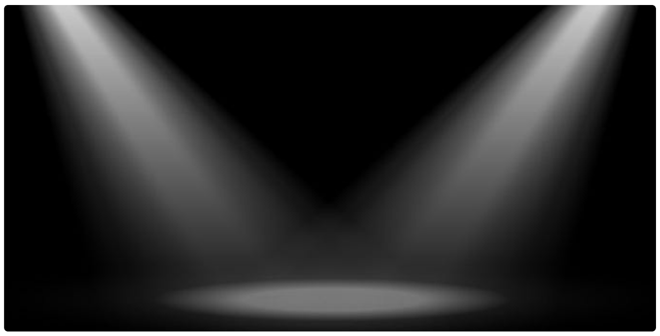
Amanatullah

Transformer Architecture explained

Transformers are a new development in machine learning that have been making a lo...

10 min read · Sep 1, 2023

358 2



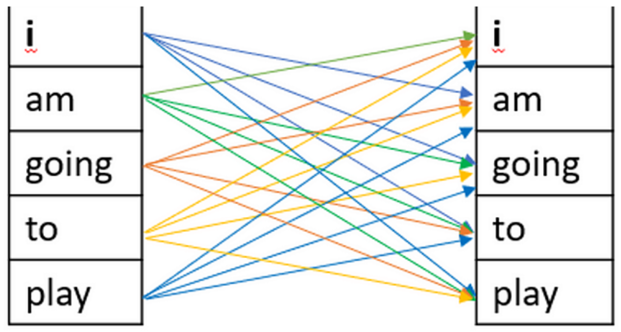
Assaad MOAWAD in DataThings

Masked multi-head attention explained in a simple way

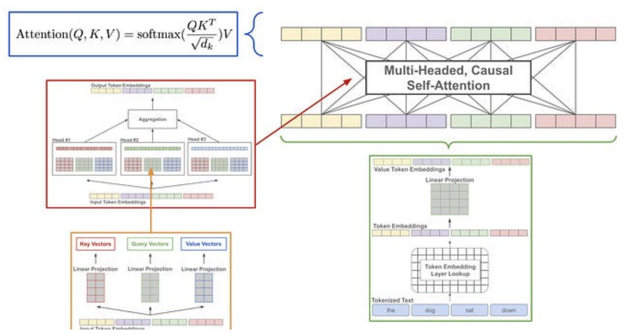
How does chat GPT actually work? What is the attention mechanism behind it?

· 12 min read · Aug 29, 2023

20



Lovelyn David



Akash Kesrwani

Self-Attention: A step-by-step guide to calculating the context...

Introduction

7 min read · Oct 16, 2023

4

Multi-Head Self Attention: Short Understanding

Each "block" of a large language model (LLM) is comprised of self-attention and a feed-...

3 min read · Sep 8, 2023

51 1

See more recommendations