Introduction to Large Language Models (LLMs)

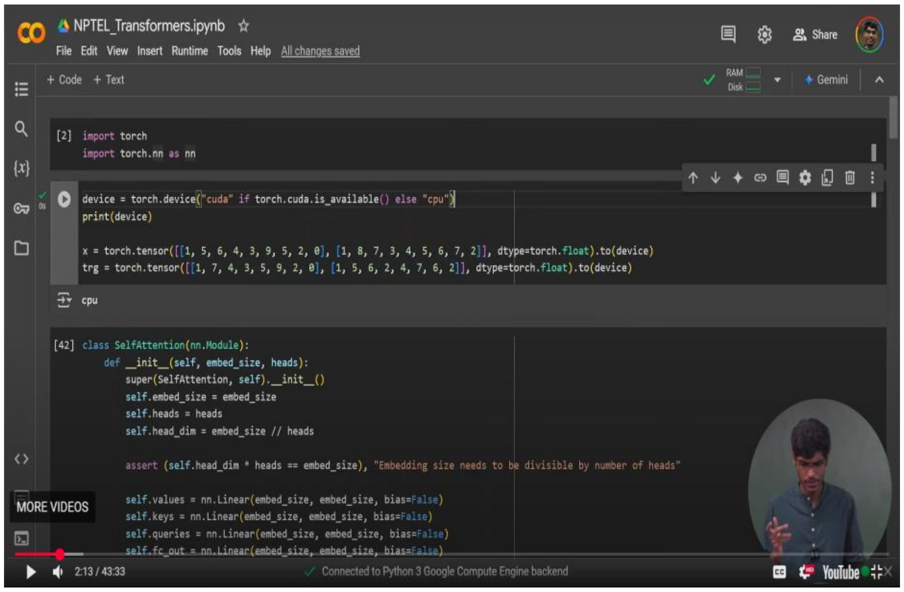
Prof. Tanmoy Chakraborty, Prof. Soumen Chakraborti

Department of Computer Science & Engineering

Indian Institute of Technology, Delhi

Lecture 17

Implementation of Transformer using PyTorch


NPTEL_Transformers.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
RAM
Disk
Gemini
Share
MORE VIDEOS
2:13 / 43:33
Connected to Python 3 Google Compute Engine backend

```
[2] import torch
    import torch.nn as nn
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)

x = torch.tensor([[1, 5, 6, 4, 3, 9, 5, 2, 0], [1, 8, 7, 3, 4, 5, 6, 7, 2]], dtype=torch.float).to(device)
trg = torch.tensor([[1, 7, 4, 3, 5, 9, 2, 0], [1, 5, 6, 2, 4, 7, 6, 2]], dtype=torch.float).to(device)
```

```
cpu
```

```
[42] class SelfAttention(nn.Module):
         def __init__(self, embed_size, heads):
             super(SelfAttention, self).__init__()
             self.embed_size = embed_size
             self.heads = heads
             self.head_dim = embed_size // heads

             assert (self.head_dim * heads == embed_size), "Embedding size needs to be divisible by number of heads"

             self.values = nn.Linear(embed_size, embed_size, bias=False)
             self.keys = nn.Linear(embed_size, embed_size, bias=False)
             self.queries = nn.Linear(embed_size, embed_size, bias=False)
             self.fc_out = nn.Linear(embed_size, embed_size, bias=False)
```

Hello, everyone. So in this session, we will be covering some implementation part. We'll be coding transformers from scratch in PyTorch. So in this week, what we saw was how transformers and the self-attention mechanism in transformer works. We saw different components of transformers like self-attention, multi-rated self-attention. Then we moved to positional encoding.

And finally, we saw how layer normalization works. So we will put all this into practice, code this up in PyTorch. In the interest of time, I have already kind of compiled all the code that we are going to show today. And I will describe step by step, every step, how we implement and things like that.

So let's start. So in the second week, we already in the third week i guess we already saw the basics of pytorch, what are tensors and how autograd works how gradient descent is done and all these things. So now we will use all of those concepts to implement transforms So for implementing, we import Torch, which is the PyTorch library and the torch.nn, which consists of all the functionalities you need to implement neural networks. As explained, device basically checks whether if CUDA, CUDA stands for basically, you can think of it as GPUs.

So if you have some GPUs or graphics processing units available in your server, then Torch will kind of search it whether it is available in your current environment. If it is not available, we can use CPU. Colab provides some hours of free GPU access. But since this is a kind of toy exercise, we're not going to do some really large transformers. We're just doing a toy transformer.

So CPU will be enough in our case. Okay. So, let's start by the self-attention module.

NPTEL_Transformers.ipynb - Cola
1706.03762
https://arxiv.org/pdf/1706.03762
Draw
1
of 15

Provided proper attribution is provided, Google hereby grants permission to reproduce the tables and figures in this paper solely for use in journalistic or scholarly works.

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer,

03762v7 [cs.CL] 2 Aug 2023

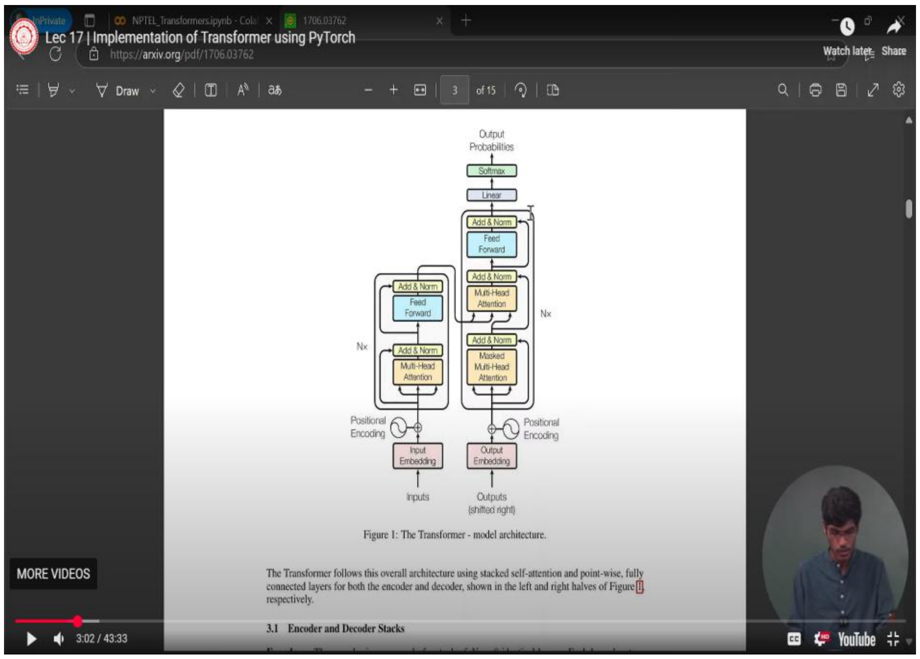


Figure 1: The Transformer - model architecture.

The Transformer follows this overall architecture using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder, shown in the left and right halves of Figure 1, respectively.

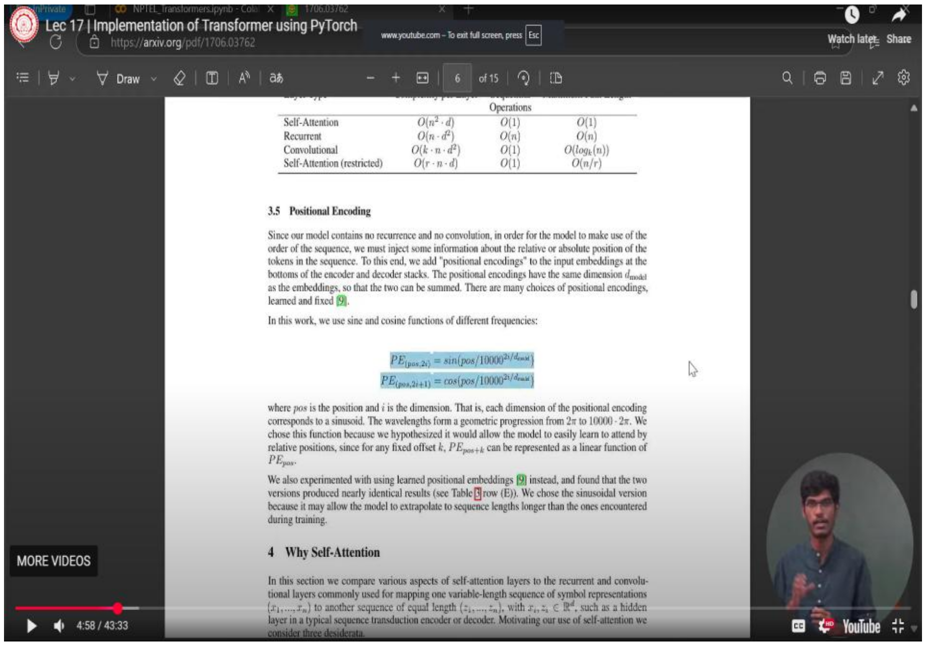
3.1 Encoder and Decoder Stacks

So, as we know, the paper which first proposed attention mechanism was in fact self-attention mechanism because attention was already previously proposed for RNNs and LSTMs and sequence-to-sequence models, basically. So, self-attention was proposed in this paper, attention is all you need, and we will be religiously following this paper for implementing transformers using PyTorch.

So, let us look at the architecture which the authors proposed. So, we have already previously seen this diagram, but let us revisit this one before moving on to coding the part. So this is an encoder-decoder architecture, in the same way what we have discussed in this week. Here, first, if you see the left-hand side block, that is an encoder block, right? So the encoder block, consists of basically three modules, if you see in that way. First is the multi-head attention module, second is the feed forward module, and third is the add and not module.

So focus on this left-handed block, right? So what it goes is, first you have inputs. So inputs are like, NPTEL is great, right? This is a text input. And then we have input embedding, which, so first, this text is tokenized. So we have already seen the tokenization algorithms, different tokenization algorithms, like byte pair encoding, like sentence-based tokenizer, word-based, all this we have seen. So, first the text is tokenized.

So, suppose NPTEL is great, we will see this in practice in subsequent weeks when we discuss in the HuggingFest library and HuggingFest tutorial, there we'll explain and show how tokenizers work basically in practice. So suppose we have this text, NPTEL is great, and for now let us assume it is tokenized as 'NPTEL' a single token, 'is' is a single token, and 'Great' is a single token. Now each token is then passed to an input embedding layer. So what input embedding layer does? It is basically kind of a lookup table, where for every token in the vocabulary, which is already present, we have an embedding. Embedding is basically a vector representation of the token, right? And then we add positional encoding.


Lec 17 | Implementation of Transformer using PyTorch
https://arxiv.org/pdf/1706.03762
www.youtube.com – To exit full screen, press Esc
Draw
6
of 15
Watch later
Share
MORE VIDEOS
4:58 / 43:33
YouTube

		Operations	
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

3.5 Positional Encoding

Since our model contains no recurrence and no convolution, in order for the model to make use of the order of the sequence, we must inject some information about the relative or absolute position of the tokens in the sequence. 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 d_{model} as the embeddings, so that the two can be summed. There are many choices of positional encodings, learned and fixed [9].

In this work, we use sine and cosine functions of different frequencies:

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

where pos is the position and i is the dimension. That is, each dimension of the positional encoding corresponds to a sinusoid. The wavelengths form a geometric progression from 2π to $10000 \cdot 2\pi$. We chose this function because we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}.

We also experimented with using learned positional embeddings [9] instead, and found that the two versions produced nearly identical results (see Table 3 row (E)). We chose the sinusoidal version because it may allow the model to extrapolate to sequence lengths longer than the ones encountered during training.

4 Why Self-Attention

In this section we compare various aspects of self-attention layers to the recurrent and convolutional layers commonly used for mapping one variable-length sequence of symbol representations $(x_1, ..., x_n)$ to another sequence of equal length $(z_1, ..., z_n)$, with $x_i, z_i \in \mathbb{R}^d$, such as a hidden layer in a typical sequence transduction encoder or decoder. Motivating our use of self-attention we consider three desiderata.

So in this paper, they proposed, as you already have seen, the sinusoidal positional encoding we see here. So the sinusoidal positional embedding, what it says is, it's sine of position by some number to the power 2i by d model. And it alternates for each dimension between sine and cos. So this is a kind of static positional embedding, static and absolute. Static in the sense it is already predefined for every dimension, every dimension i and every position pos, pos and every dimension i, it is already predefined.

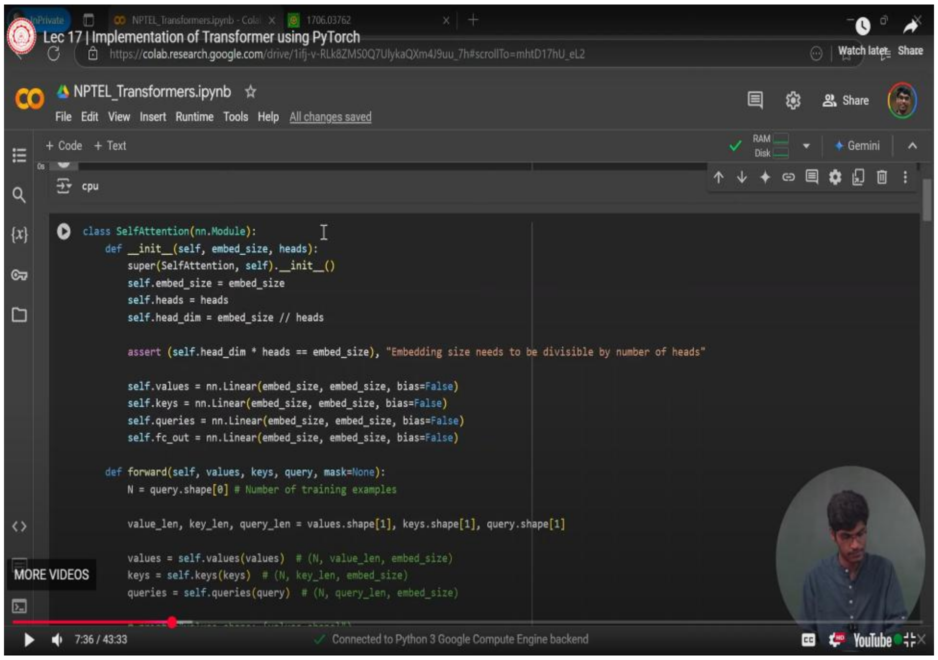
It is not trained explicitly. And it is absolute in the sense it considers the absolute position of the token, whether it is at 0 index, whether it is 1 index, and like that. So it is not relative. So gradually, we saw how there are propositions on trainable positional encodings, which are dynamically updated during training, and also relative positional encodings, which instead of taking the absolute position 0,1, it takes the relative position between the tokens. So we have already seen all this in this week.

So this multi-head attention. So takes in three inputs the query vector or the query matrix the key matrix and the value matrix. For the encoder part all the query key and value comes from the input right whatever the input we are giving after getting the input embedding additional the positional encoding we have say vector matrix x. So q k v query key and value matrix all are coming from the same input matrix. Then multi-head attention gives whatever it does.

We will see that again. Then it is add or none. What it does is there's a residual connection. So the input matrix is again added to the output of the multi-head attention. We apply a near-none on top of it, and then pass it onto a feed-forward network.

So this is an encoder block, a single encoder block. And it is stacked n times. So there are n encoder layers. And then we have the decoder, which we'll see next. So what is the main part of this? The main tough part where you need to do a lot of thinking how to implement it is basically the multi-head attention part.

This is multi-head self-attention. So we will see this first, this module first.


Lec 17 | Implementation of Transformer using PyTorch
NPTEL_Transformers.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Gemini
MORE VIDEOS
7:36 / 43:33

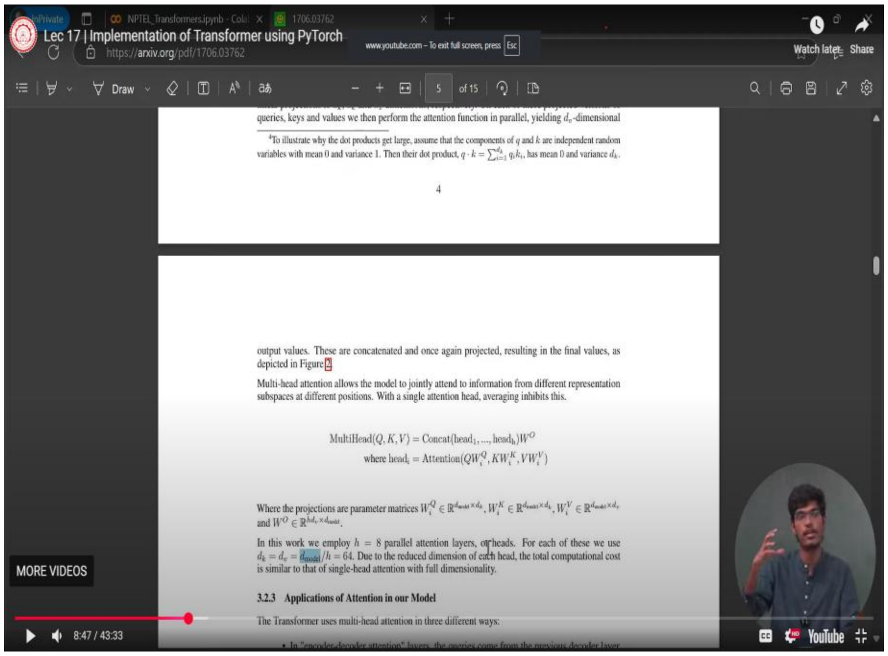
```
cpu
```

```python
class SelfAttention(nn.Module):
    def __init__(self, embed_size, heads):
        super(SelfAttention, self).__init__()
        self.embed_size = embed_size
        self.heads = heads
        self.head_dim = embed_size // heads

        assert (self.head_dim * heads == embed_size), "Embedding size needs to be divisible by number of heads"

        self.values = nn.Linear(embed_size, embed_size, bias=False)
        self.keys = nn.Linear(embed_size, embed_size, bias=False)
        self.queries = nn.Linear(embed_size, embed_size, bias=False)
        self.fc_out = nn.Linear(embed_size, embed_size, bias=False)

    def forward(self, values, keys, query, mask=None):
        N = query.shape[0] # Number of training examples

        value_len, key_len, query_len = values.shape[1], keys.shape[1], query.shape[1]

        values = self.values(values)  # (N, value_len, embed_size)
        keys = self.keys(keys)  # (N, key_len, embed_size)
        queries = self.queries(query)  # (N, query_len, embed_size)
```

Connected to Python 3 Google Compute Engine backend

So this class self-attention implements that multi-headed self-attention part. So for any class, hopefully you have already done the prerequisites for Python and object-oriented programming. So you must know that whenever we declare any class in any object-oriented programming language, be it C++, be it Python, be it Java, we need a class constructor, right? So the class constructor basically defines how we're going to handle the different attributes of the object which will be instantiated from the class.

So this init method is basically, it instantiates the class. It is a constructor method. So our embed size is basically the model embedding size.


Lec 17 | Implementation of Transformer using PyTorch
https://arxiv.org/pdf/1706.03762
www.youtube.com – To exit full screen, press Esc
Draw
5 of 15
Watch later
Share
MORE VIDEOS
8:47 / 43:33

queries, keys and values we then perform the attention function in parallel, yielding d_v-dimensional

[4]To illustrate why the dot products get large, assume that the components of q and k are independent random variables with mean 0 and variance 1. Then their dot product, $q \cdot k = \sum_{i=1}^{d_k} q_i k_i$, has mean 0 and variance d_k.

4

output values. These are concatenated and once again projected, resulting in the final values, as depicted in Figure 2.

Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this.

$$\mathrm{MultiHead}(Q,K,V) = \mathrm{Concat}(\mathrm{head}_1, ..., \mathrm{head}_h)W^O$$
$$\text{where } \mathrm{head}_i = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{model}}$.

In this work we employ $h = 8$ parallel attention layers, or heads. For each of these we use $d_k = d_v = d_{model}/h = 64$. Due to the reduced dimension of each head, the total computational cost is similar to that of single-head attention with full dimensionality.

3.2.3 Applications of Attention in our Model

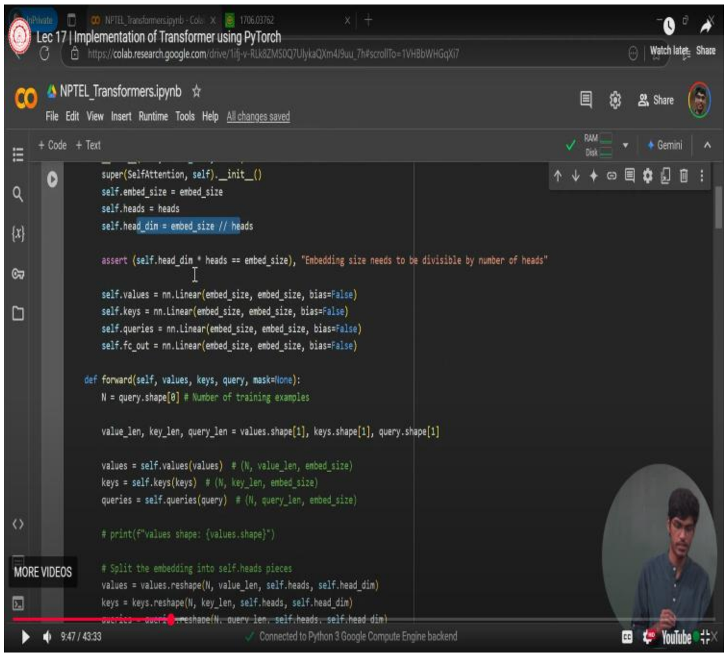
The Transformer uses multi-head attention in three different ways:

So if we want to draw parallels with the paper, if we see here, so let's see the multi-head self-attention section. So here you can see that this D model, this D model is basically the embedding dimension.

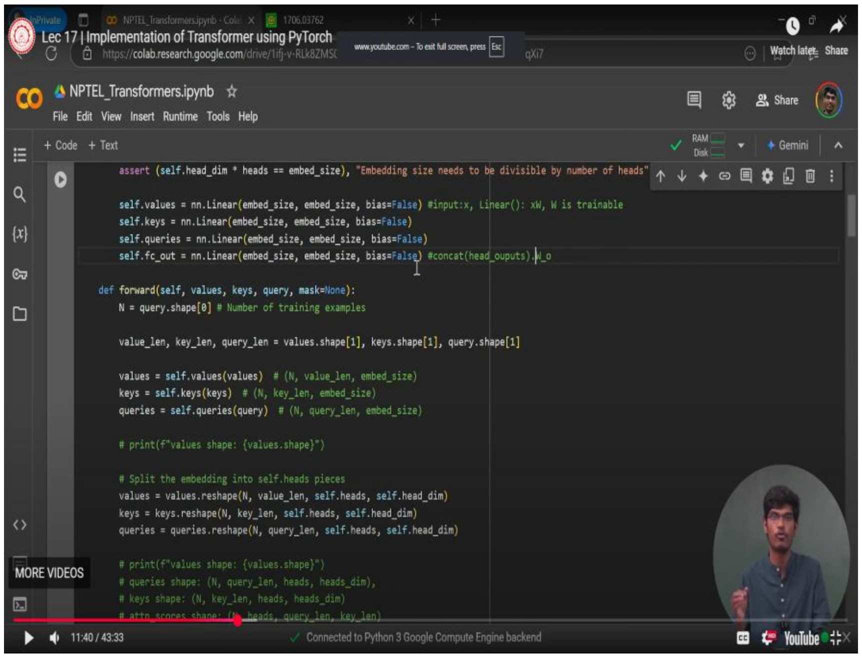
And this dimension is consistent throughout the model. That is, each layer output is of dimension D model. So this D model we define as our embed size. And heads is the number of attention heads at each layer. So what will be the head dimension? If we see here, so it says that dk and dv, so dk and dv, dk is the dimension of the key and the query.

So the key and the query vectors, or query stack vectors, they become matrices. The key and the query matrices always have the same embedding dimension, right? So this dq and dk are basically the same. So dk is the key dimension, and dv is the dimension of the value vectors. So what is it? It is D model by H. So D model is our embed size and H is the number of heads.

So that's what we do. Head dimension is embed size by number of heads.


Lec 17 | Implementation of Transformer using PyTorch
NPTEL_Transformers.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
MORE VIDEOS
9:47 / 43:33
Connected to Python 3 Google Compute Engine backend

```
        super(SelfAttention, self).__init__()
        self.embed_size = embed_size
        self.heads = heads
        self.head_dim = embed_size // heads

        assert (self.head_dim * heads == embed_size), "Embedding size needs to be divisible by number of heads"

        self.values = nn.Linear(embed_size, embed_size, bias=False)
        self.keys = nn.Linear(embed_size, embed_size, bias=False)
        self.queries = nn.Linear(embed_size, embed_size, bias=False)
        self.fc_out = nn.Linear(embed_size, embed_size, bias=False)

    def forward(self, values, keys, query, mask=None):
        N = query.shape[0] # Number of training examples

        value_len, key_len, query_len = values.shape[1], keys.shape[1], query.shape[1]

        values = self.values(values)  # (N, value_len, embed_size)
        keys = self.keys(keys)  # (N, key_len, embed_size)
        queries = self.queries(query)  # (N, query_len, embed_size)

        # print(f"values shape: {values.shape}")

        # Split the embedding into self.heads pieces
        values = values.reshape(N, value_len, self.heads, self.head_dim)
        keys = keys.reshape(N, key_len, self.heads, self.head_dim)
```


Lec 17 | Implementation of Transformer using PyTorch
NPTEL_Transformers.ipynb
File Edit View Insert Runtime Tools Help
+ Code + Text
MORE VIDEOS
11:40 / 43:33
Connected to Python 3 Google Compute Engine backend

```
        assert (self.head_dim * heads == embed_size), "Embedding size needs to be divisible by number of heads"

        self.values = nn.Linear(embed_size, embed_size, bias=False) #input:x, Linear(): xW, W is trainable
        self.keys = nn.Linear(embed_size, embed_size, bias=False)
        self.queries = nn.Linear(embed_size, embed_size, bias=False)
        self.fc_out = nn.Linear(embed_size, embed_size, bias=False) #concat(head_ouputs).W_o

    def forward(self, values, keys, query, mask=None):
        N = query.shape[0] # Number of training examples

        value_len, key_len, query_len = values.shape[1], keys.shape[1], query.shape[1]

        values = self.values(values)  # (N, value_len, embed_size)
        keys = self.keys(keys)  # (N, key_len, embed_size)
        queries = self.queries(query)  # (N, query_len, embed_size)

        # print(f"values shape: {values.shape}")

        # Split the embedding into self.heads pieces
        values = values.reshape(N, value_len, self.heads, self.head_dim)
        keys = keys.reshape(N, key_len, self.heads, self.head_dim)
        queries = queries.reshape(N, query_len, self.heads, self.head_dim)

        # print(f"values shape: {values.shape}")
        # queries shape: (N, query_len, heads, heads_dim),
        # keys shape: (N, key_len, heads, heads_dim)
        # attn_scores shape: (N, heads, query_len, key_len)
```

And we write a condition, an assertion condition to check that the embedding size, the embedding dimension should always be divisible by the number of heads. So when you multiply the number of heads and our head dimension, we should get the embed dimension. This is a kind of assertion. And we define three layers, linear layers.

Linear layers means simple linear transformation, multiplying a projection matrix. So if you have Kx, it is just writing Wx. So that is, W is our learnable matrix. So that's what linear layer does, right? So maybe I can write here. So what linear layer does is, if you have an input, say x, then after passing on this linear, right what it will does is it will just make it wx right.

So this w where w is trainable right w is a trainable matrix and we define such linear layers for every values keys and queries and this fcout is basically where we multiply wo if you see here after see after concatenation we multiply another matrix wo right So that's what this fcout does. You can say it as wx or in our case, it's basically xw. So it's xw. So this fcout is basically then wo. So whatever after concatting the head outputs, we apply wo on top of it.

This is what fcout does. Now let's look at how to define the forward pass through the self-attention. So as I said, what is the input to the multi-headed self-attention module? It's three matrices, the key matrix, the query matrix, and the value matrix. So we define n as the first dimension of the query matrix. So what is the first dimension of the query matrix? The query matrix if we will see in details there later. So the query matrix whatever we have this query is basically of shape.

Shape is basically the dimensions of the matrix right. So first is n. so what is n? n is the number of training examples number of examples in a batch right. And then we have how many tokens are there in the query.

So we can define it as query len right. We can define it as query len then we have the embedding dimension right embedding size in our case so this is a query how it is going in and so okay, after that, what do we do?

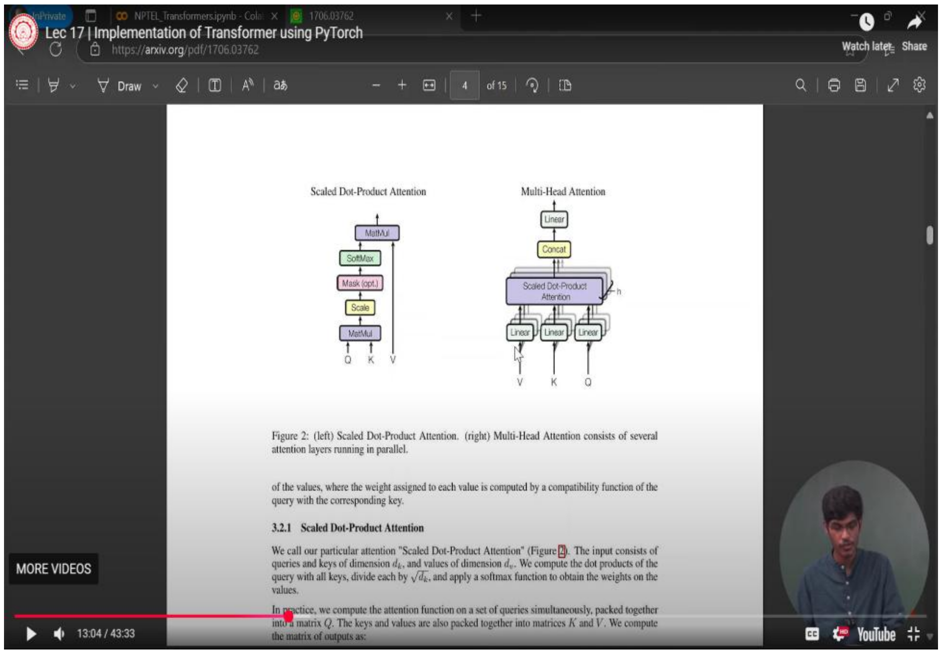

Lec 17 | Implementation of Transformer using PyTorch
https://arxiv.org/pdf/1706.03762
Scaled Dot-Product Attention
MatMul
SoftMax
Mask (opt.)
Scale
MatMul
Q
K
V
Multi-Head Attention
Linear
Concat
Scaled Dot-Product Attention
h
Linear
Linear
Linear
V
K
Q
MORE VIDEOS
13:04 / 43:33

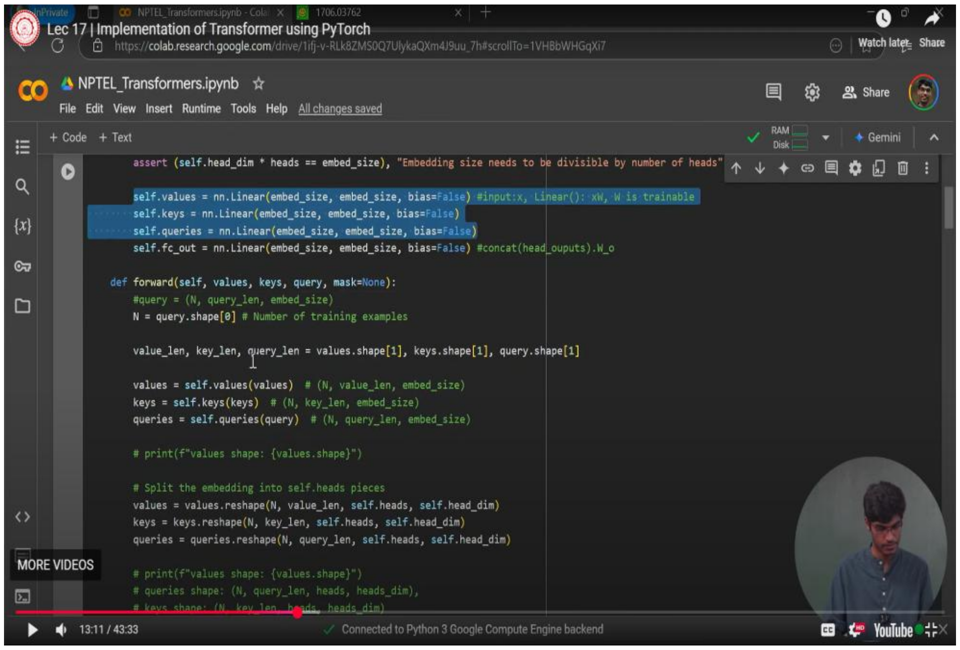
Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

3.2.1 Scaled Dot-Product Attention

We call our particular attention "Scaled Dot-Product Attention" (Figure 2). The input consists of queries and keys of dimension d_k, and values of dimension d_v. We compute the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix Q. The keys and values are also packed together into matrices K and V. We compute the matrix of outputs as:


Lec 17 | Implementation of Transformer using PyTorch
https://colab.research.google.com/drive/1ifj-v-RLk8ZMS0Q7UlykaQXm4J9uu_7h#scrollTo=1VHBbWHGqXi7
NPTEL_Transformers.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Share
Gemini
MORE VIDEOS
13:11 / 43:33
Connected to Python 3 Google Compute Engine backend

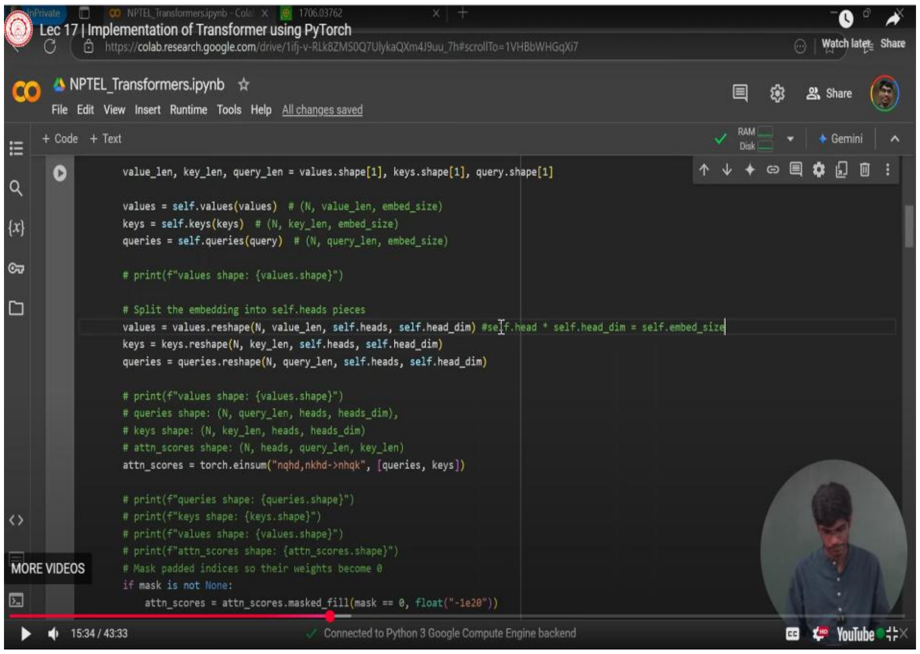
```python
        assert (self.head_dim * heads == embed_size), "Embedding size needs to be divisible by number of heads"

        self.values = nn.Linear(embed_size, embed_size, bias=False) #input:x, Linear(): xW, W is trainable
        self.keys = nn.Linear(embed_size, embed_size, bias=False)
        self.queries = nn.Linear(embed_size, embed_size, bias=False)
        self.fc_out = nn.Linear(embed_size, embed_size, bias=False) #concat(head_ouputs).W_o

    def forward(self, values, keys, query, mask=None):
        #query = (N, query_len, embed_size)
        N = query.shape[0] # Number of training examples

        value_len, key_len, query_len = values.shape[1], keys.shape[1], query.shape[1]

        values = self.values(values)  # (N, value_len, embed_size)
        keys = self.keys(keys)  # (N, key_len, embed_size)
        queries = self.queries(query)  # (N, query_len, embed_size)

        # print(f"values shape: {values.shape}")

        # Split the embedding into self.heads pieces
        values = values.reshape(N, value_len, self.heads, self.head_dim)
        keys = keys.reshape(N, key_len, self.heads, self.head_dim)
        queries = queries.reshape(N, query_len, self.heads, self.head_dim)

        # print(f"values shape: {values.shape}")
        # queries shape: (N, query_len, heads, heads_dim),
        # keys shape: (N, key_len, heads, heads_dim)
```

Look, so you have the QKV, you pass it through a linear layer, each of the matrices. So these linear layers are defined here, these three linear layers, right? And then we have n which is the number of training examples in the batch, and this value lane key layer and query lane are basically the number of tokens in the query matrix the key matrix and the value matrix. So number of tokens is defined as basically this query length what we are saying right. So now because look in the encoder case the query length key length and value length are the same. Because all the query key and value comes from the inputs.

In case of decoder on the other hand the query comes from the input to the decoder but the key and values come from the encoder module right. So the key and the value may not have the same length as the or rather in most of the times don't have the same length as the query. So that's how we handle it explicitly. And when we have this key querying values as n which is the number of training examples, say len which is the number of tokens and embed size. So this embed size is now basically if you think of in that way this is heads number of heads into head dimension which you already saw.

So we need to break this up into different heads, so we have already multiplied it by the projection, now we divide it into different heads explicitly right So we do that here. What we do is we accept the thought there is this concept of reset, resetting tensors. So we keep the first two dimensions same. Right. It's n and value layer, key length, whatever it is.

And we divide the embed size into number of heads and the dimension of each head. So the multiplication of this last two dimension.


Lec 17 | Implementation of Transformer using PyTorch
NPTEL_Transformers.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
MORE VIDEOS
15:34 / 43:33
Connected to Python 3 Google Compute Engine backend

```
value_len, key_len, query_len = values.shape[1], keys.shape[1], query.shape[1]

values = self.values(values)  # (N, value_len, embed_size)
keys = self.keys(keys)  # (N, key_len, embed_size)
queries = self.queries(query)  # (N, query_len, embed_size)

# print(f"values shape: {values.shape}")

# Split the embedding into self.heads pieces
values = values.reshape(N, value_len, self.heads, self.head_dim) #self.head * self.head_dim = self.embed_size
keys = keys.reshape(N, key_len, self.heads, self.head_dim)
queries = queries.reshape(N, query_len, self.heads, self.head_dim)

# print(f"values shape: {values.shape}")
# queries shape: (N, query_len, heads, heads_dim),
# keys shape: (N, key_len, heads, heads_dim)
# attn_scores shape: (N, heads, query_len, key_len)
attn_scores = torch.einsum("nqhd,nkhd->nhqk", [queries, keys])

# print(f"queries shape: {queries.shape}")
# print(f"keys shape: {keys.shape}")
# print(f"values shape: {values.shape}")
# print(f"attn_scores shape: {attn_scores.shape}")
# Mask padded indices so their weights become 0
if mask is not None:
    attn_scores = attn_scores.masked_fill(mask == 0, float("-1e20"))
```

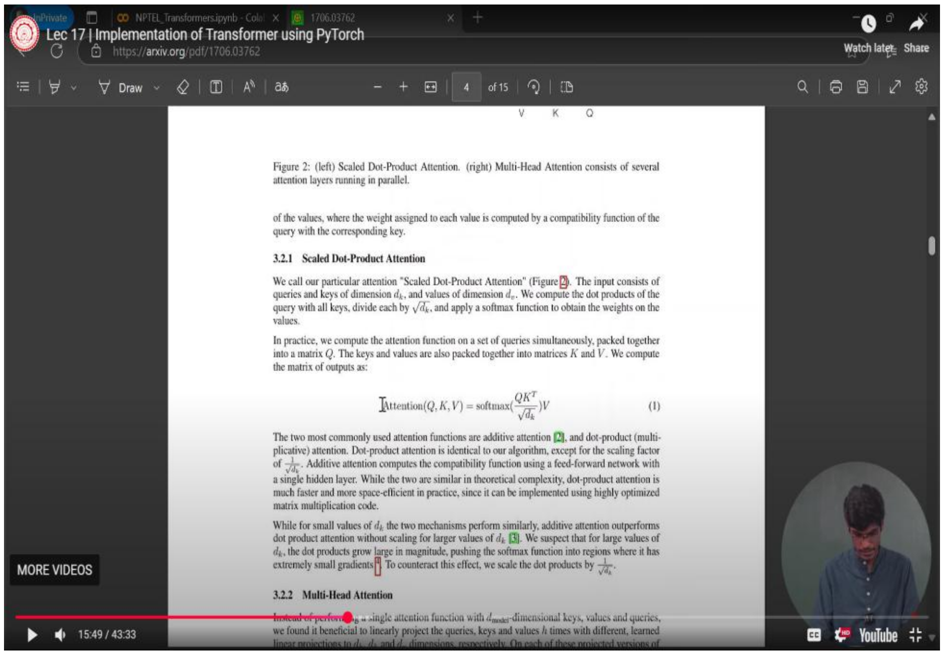

Lec 17 | Implementation of Transformer using PyTorch

Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

3.2.1 Scaled Dot-Product Attention

We call our particular attention "Scaled Dot-Product Attention" (Figure 2). The input consists of queries and keys of dimension d_k, and values of dimension d_v. We compute the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix Q. The keys and values are also packed together into matrices K and V. We compute the matrix of outputs as:

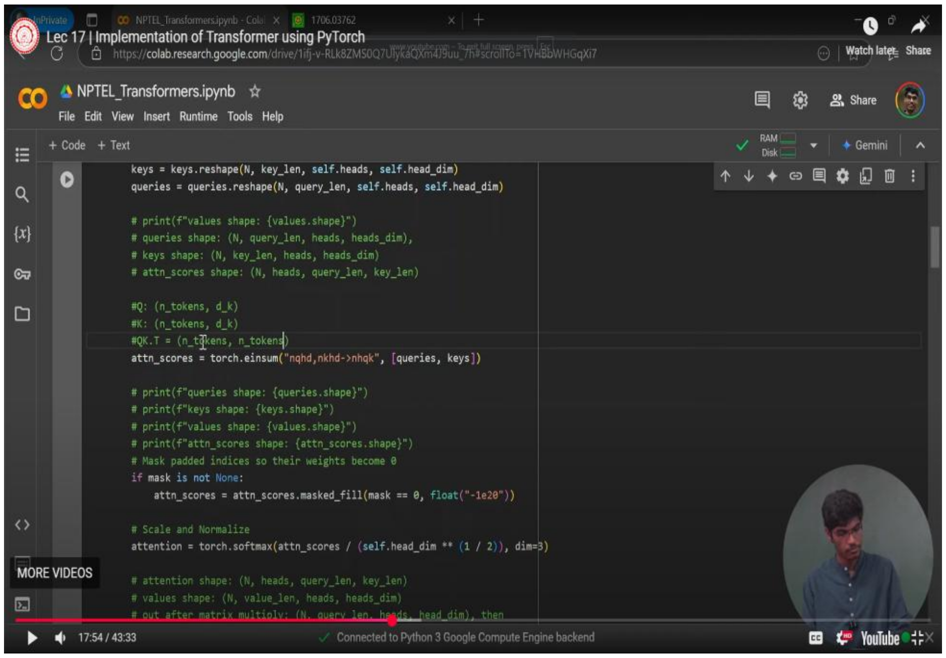
$$\mathrm{Attention}(Q,K,V) = \mathrm{softmax}(\frac{QK^T}{\sqrt{d_k}})V \quad (1)$$

The two most commonly used attention functions are additive attention [2], and dot-product (multiplicative) attention. Dot-product attention is identical to our algorithm, except for the scaling factor of $\frac{1}{\sqrt{d_k}}$. Additive attention computes the compatibility function using a feed-forward network with a single hidden layer. While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.

While for small values of d_k the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of d_k [3]. We suspect that for large values of d_k, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients [4]. To counteract this effect, we scale the dot products by $\frac{1}{\sqrt{d_k}}$.

3.2.2 Multi-Head Attention

Instead of performing a single attention function with d_{model}-dimensional keys, values and queries, we found it beneficial to linearly project the queries, keys and values h times with different, learned

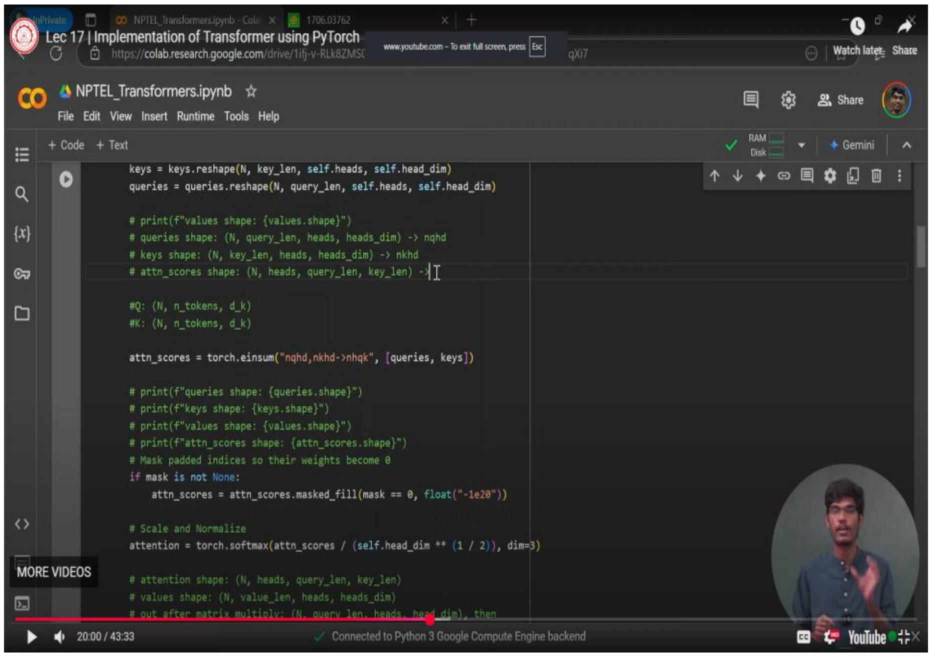

Lec 17 | Implementation of Transformer using PyTorch

```python
        keys = keys.reshape(N, key_len, self.heads, self.head_dim)
        queries = queries.reshape(N, query_len, self.heads, self.head_dim)

        # print(f"values shape: {values.shape}")
        # queries shape: (N, query_len, heads, heads_dim),
        # keys shape: (N, key_len, heads, heads_dim)
        # attn_scores shape: (N, heads, query_len, key_len)

        #Q: (n_tokens, d_k)
        #K: (n_tokens, d_k)
        #QK.T = (n_tokens, n_tokens)
        attn_scores = torch.einsum("nqhd,nkhd->nhqk", [queries, keys])

        # print(f"queries shape: {queries.shape}")
        # print(f"keys shape: {keys.shape}")
        # print(f"values shape: {values.shape}")
        # print(f"attn_scores shape: {attn_scores.shape}")
        # Mask padded indices so their weights become 0
        if mask is not None:
            attn_scores = attn_scores.masked_fill(mask == 0, float("-1e20"))

        # Scale and Normalize
        attention = torch.softmax(attn_scores / (self.head_dim ** (1 / 2)), dim=3)

        # attention shape: (N, heads, query_len, key_len)
        # values shape: (N, value_len, heads, heads_dim)
        # out after matrix multiply: (N, query_len, heads, head_dim), then
```

Connected to Python 3 Google Compute Engine backend

So we already asserted the self.heads into self.headDimension which is the dimension of the individual heads is equal to self.embedSides. We already made that assertion, right? So now we have the key query and the value matrices passed through the linear layer. So it's different for every heads, right? Now, how do you calculate the attention scores? So, this is how we calculate the attention scores. So, this part we call attention scores. So, it is Q K transpose by scaled by the square root of the dimension of the keys.

So, let's first focus one by one. So, first is Q K transpose. Now, let's think. When you have a normal query and key vectors right, where Q is basically say your number of tokens and DK right, and your K matrix is also the number of tokens into dk, number of tokens comma dk basically. So if these are your shape of q and k then it's pretty easy to do q k transpose right q k, let me write q k dot t so this is q k transpose and... So we already have, suppose we have the Q matrix of which is just basically think of it as a single example right a single sentence. So Q matrix is of say n tokens is the number of tokens comma the dimension of the key vector because Q and K have the same dimensions and the K vector or matrix whatever is simply the number of tokens and comma the dimension of the key. Then we can do simple QK transpose, which will give us basically our attention map or attention scores of a square matrix with dimension, number of tokens, common number of tokens. But when we have this batch, when we add this N over here, then QK transpose doesn't work as intended. So now what do we do?


Lec 17 | Implementation of Transformer using PyTorch
https://colab.research.google.com/drive/1ifj-v-RLk8ZMSC
NPTEL_Transformers.ipynb
File Edit View Insert Runtime Tools Help
+ Code + Text
Share
Gemini
MORE VIDEOS
20:00 / 43:33
Connected to Python 3 Google Compute Engine backend

```
keys = keys.reshape(N, key_len, self.heads, self.head_dim)
queries = queries.reshape(N, query_len, self.heads, self.head_dim)

# print(f"values shape: {values.shape}")
# queries shape: (N, query_len, heads, heads_dim) -> nqhd
# keys shape: (N, key_len, heads, heads_dim) -> nkhd
# attn_scores shape: (N, heads, query_len, key_len) ->

#Q: (N, n_tokens, d_k)
#K: (N, n_tokens, d_k)

attn_scores = torch.einsum("nqhd,nkhd->nhqk", [queries, keys])

# print(f"queries shape: {queries.shape}")
# print(f"keys shape: {keys.shape}")
# print(f"values shape: {values.shape}")
# print(f"attn_scores shape: {attn_scores.shape}")
# Mask padded indices so their weights become 0
if mask is not None:
    attn_scores = attn_scores.masked_fill(mask == 0, float("-1e20"))

# Scale and Normalize
attention = torch.softmax(attn_scores / (self.head_dim ** (1 / 2)), dim=3)

# attention shape: (N, heads, query_len, key_len)
# values shape: (N, value_len, heads, heads_dim)
# out after matrix multiply: (N, query_len, heads, head_dim), then
```

So there is this method called INSUM.

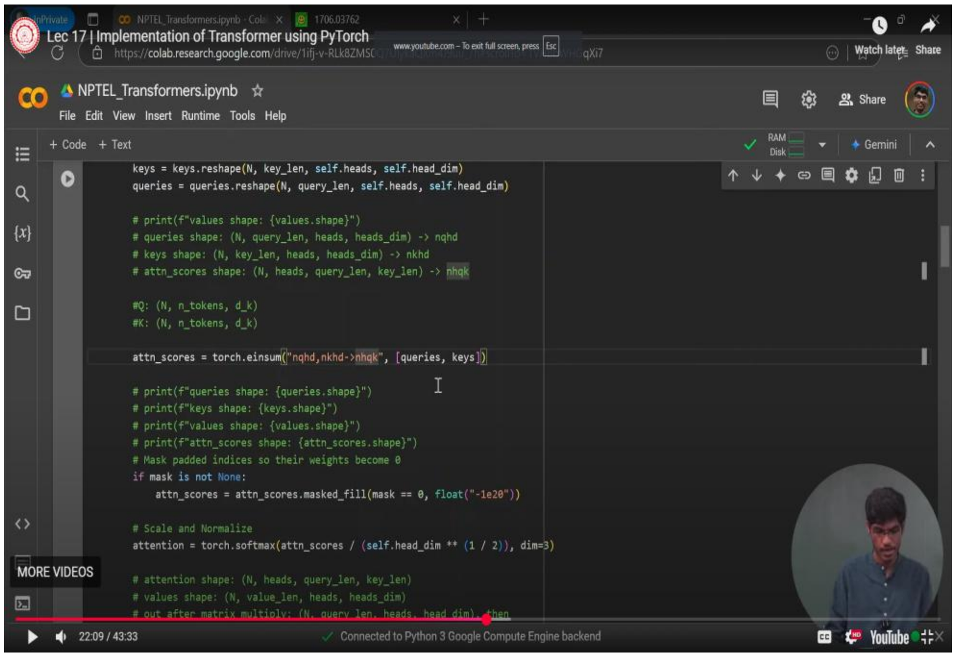
So INSUM is basically a more generalized function for doing matrix multiplication. So what you give as input to the INSUM are your input matrices and as a rule you explicitly tell what are the dimensions of your input and to what dimension do you want it to map to?

So this is a more generalized matrix multiplication. So for example, here n is our number of sentences in our batch.

Q is the query length. Q is the query length. h is the number of heads and d is our head dimension right. So tell me what is the shape of the query vector? it's n right then the query length q then the number of heads just look here n, query length q, number of heads h, and the head dimension d. So this is how we represent the dimension of the query vector when passing it to ion sum. Similarly, tell me what is the dimension of keys? It's n, the number of training examples.

Key length is different from query length. It can be different in a generalized case. So let us call it k. And the number of heads are the same h, and the head dimension is same d.

So it's nk hd. This is the dimension of key matrix. and we want to map it to attention scores.

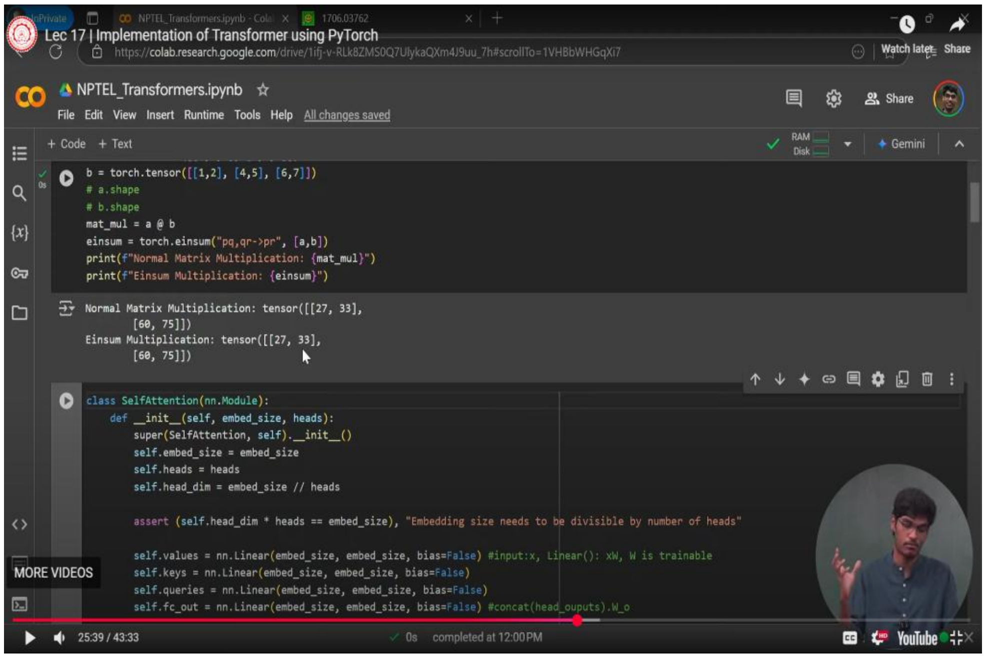

Lec 17 | Implementation of Transformer using PyTorch
NPTEL_Transformers.ipynb
File Edit View Insert Runtime Tools Help
+ Code + Text
Watch later
Share
Gemini
MORE VIDEOS
22:09 / 43:33
Connected to Python 3 Google Compute Engine backend

```
keys = keys.reshape(N, key_len, self.heads, self.head_dim)
queries = queries.reshape(N, query_len, self.heads, self.head_dim)

# print(f"values shape: {values.shape}")
# queries shape: (N, query_len, heads, heads_dim) -> nqhd
# keys shape: (N, key_len, heads, heads_dim) -> nkhd
# attn_scores shape: (N, heads, query_len, key_len) -> nhqk

#Q: (N, n_tokens, d_k)
#K: (N, n_tokens, d_k)

attn_scores = torch.einsum("nqhd,nkhd->nhqk", [queries, keys])

# print(f"queries shape: {queries.shape}")
# print(f"keys shape: {keys.shape}")
# print(f"values shape: {values.shape}")
# print(f"attn_scores shape: {attn_scores.shape}")
# Mask padded indices so their weights become 0
if mask is not None:
    attn_scores = attn_scores.masked_fill(mask == 0, float("-1e20"))

# Scale and Normalize
attention = torch.softmax(attn_scores / (self.head_dim ** (1 / 2)), dim=3)

# attention shape: (N, heads, query_len, key_len)
# values shape: (N, value_len, heads, heads_dim)
# out after matrix multiply: (N, query_len, heads, head_dim), then
```

So what should the attention scores give? For every sentence in, we should have, so for every sentence and for every head, for every sentence and for every head, we have an attention map. So ponder upon this very carefully that when we are giving a batch of sentences as input, in a multirate self-attention, at the first step what do we have? We have a square matrix. So the square matrix the y-axis in the y-axis we have the keys. So NPTEL is great right so for every token say NPTEL right, we have a attention scores over each token in the sentence.

So for NPTEL token as query the all the three acts as keys right. So we have attention score for 'NPTEL', we have attention for 'is' we have attention score for 'great' right, say it is 0.5. So before normalization before converting to probability it can be anything after converting to probability when it con when we apply short max it becomes something like say 0.5 0.3 0.2 just like that So then what is the dimensions of the attention score matrix? For every n number of training examples, for every head, we have a matrix where the number of rows is the y-axis, which is the query length. And for each query, we have attention distribution over the keys. So it's n, h, heads, q, and k. So this is a mapping we can do.

So now INSUM tells what do you give as input to the INSUM. If you give the input 1, the dimension of input 1, the dimension of input 2, give an arrow like that and tell to what you want to map it to. So we need to map it to NHQK as we discussed here. And the inputs are queries and keys.


Lec 17 | Implementation of Transformer using PyTorch
NPTEL_Transformers.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Share
Gemini
MORE VIDEOS
25:39 / 43:33
0s completed at 12:00 PM
YouTube

```
b = torch.tensor([[1,2], [4,5], [6,7]])
# a.shape
# b.shape
mat_mul = a @ b
einsum = torch.einsum("pq,qr->pr", [a,b])
print(f"Normal Matrix Multiplication: {mat_mul}")
print(f"Einsum Multiplication: {einsum}")
```

```
Normal Matrix Multiplication: tensor([[27, 33],
        [60, 75]])
Einsum Multiplication: tensor([[27, 33],
        [60, 75]])
```

```
class SelfAttention(nn.Module):
    def __init__(self, embed_size, heads):
        super(SelfAttention, self).__init__()
        self.embed_size = embed_size
        self.heads = heads
        self.head_dim = embed_size // heads

        assert (self.head_dim * heads == embed_size), "Embedding size needs to be divisible by number of heads"

        self.values = nn.Linear(embed_size, embed_size, bias=False) #input:x, Linear(): xW, W is trainable
        self.keys = nn.Linear(embed_size, embed_size, bias=False)
        self.queries = nn.Linear(embed_size, embed_size, bias=False)
        self.fc_out = nn.Linear(embed_size, embed_size, bias=False) #concat(head_ouputs).W_o
```

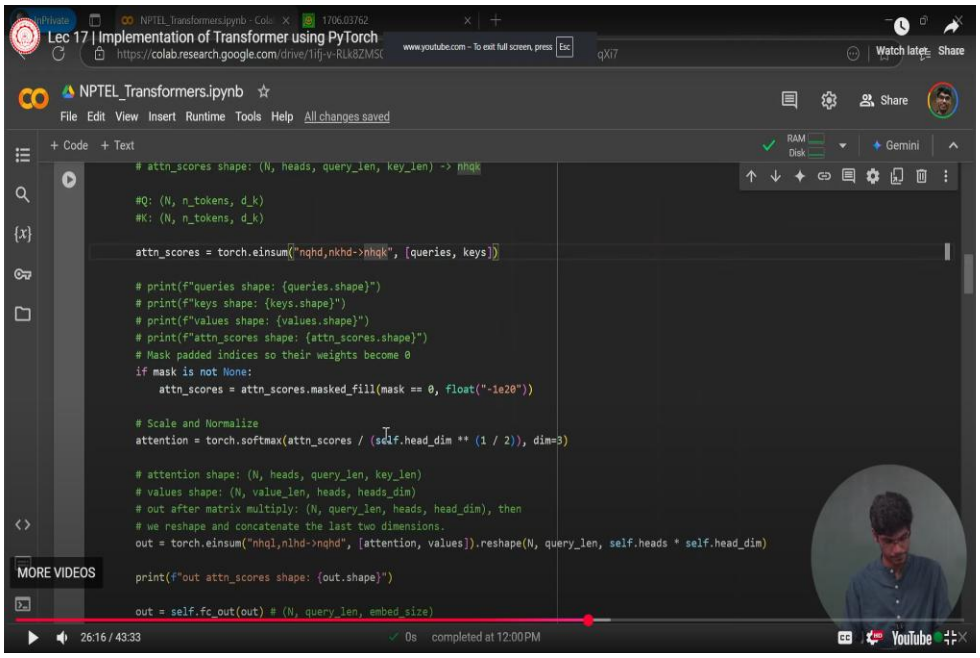
So maybe I can illustrate a little bit to give you an example that ion sum occurs in similar way to matrix multiplication.

Suppose I have torch dot tensor. Let us take a one two three right and say four five six right so this is basically a so what is a then you see a is a tensor and what is the shape of it then shape of a is so it's two comma three right two comma three um dimensional tensor Now we have, say, a B. So let us copy this. OK. So define another tensor B. And it is something 3 comma, right? So it should be, say, 1, 2, 4, 5, and 6, 7.

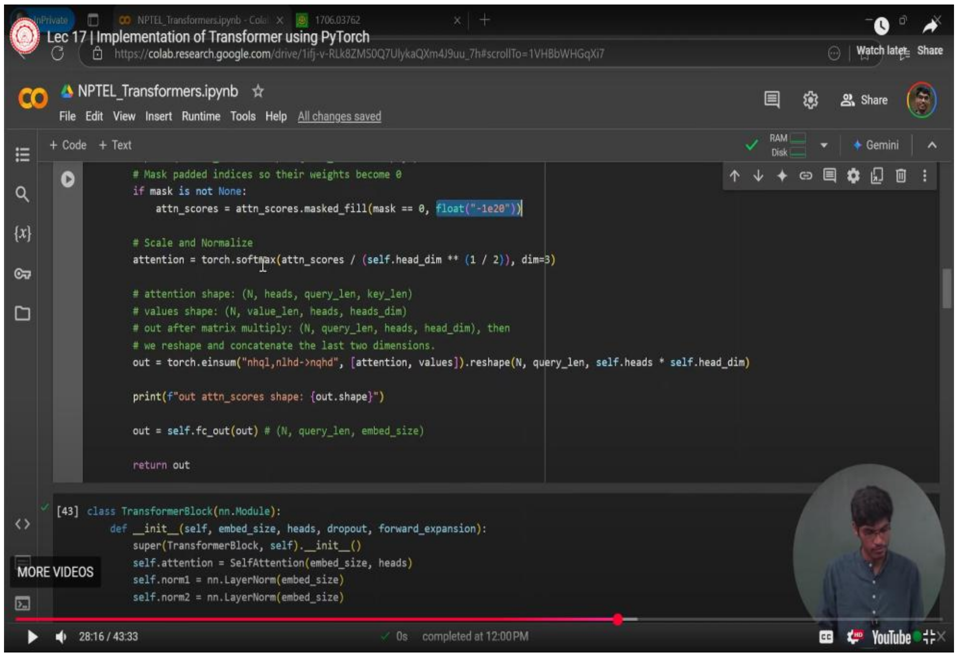
So what does that say for B? That say for B is 3 comma 2. Okay so now a and b are fit for multiplication and upper multiplication what should the dimension? 2 comma 2 So if we do a simple matrix multiplication so this is basically matmul let it let us call it matmul so this gives us some basically So let us print that too. So we can print, say, normal matrix multiplication. Now suppose we want to do ion sum.

So let us say ion sum. So in ion sum, what will you do? Tell me. So you should do things along with me you should also have a laptop and open your basically collab and just do with me okay so when you answer what do you want to give tell me we already discussed so we have a rule right a rule what is the mapping so what is the dimension of a say it is pq And what is the dimension of B? So the number of columns in A must match with the number of rows in B. So say it's QR. And what is the mapping rule? V1, PR. So this is the mapping rule in case of matrix multiplication, right? And what are the inputs? It's A, B, right? Similar to what we did here. Correct? And now, let us check whether if we do ion sum multiplication, and if we do matrix multiplication, are the results same? Let us check.

So you can check, you can see by normal matrix multiplication and by ion sum, we get the same output, right? So this is a kind of a small illustration on how ion sum works. Einsum can be helpful for dealing with more generalized, more complicated things, but this is just a kind of a toy example to illustrate that.

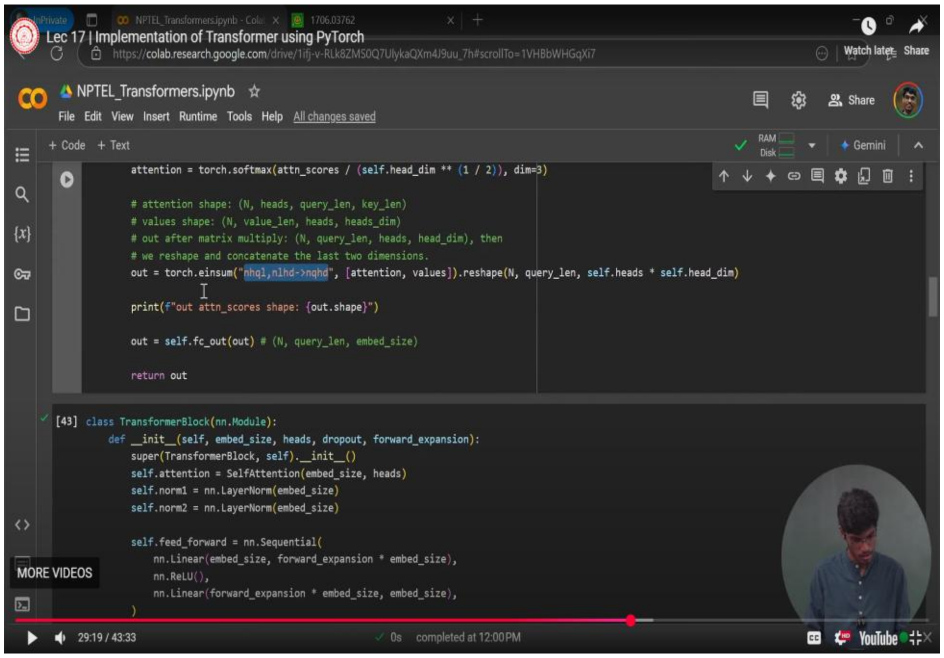

Lec 17 | Implementation of Transformer using PyTorch
NPTEL_Transformers.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
MORE VIDEOS
26:16 / 43:33
0s completed at 12:00 PM

```
# attn_scores shape: (N, heads, query_len, key_len) -> nhqk

#Q: (N, n_tokens, d_k)
#K: (N, n_tokens, d_k)

attn_scores = torch.einsum("nqhd,nkhd->nhqk", [queries, keys])

# print(f"queries shape: {queries.shape}")
# print(f"keys shape: {keys.shape}")
# print(f"values shape: {values.shape}")
# print(f"attn_scores shape: {attn_scores.shape}")
# Mask padded indices so their weights become 0
if mask is not None:
    attn_scores = attn_scores.masked_fill(mask == 0, float("-1e20"))

# Scale and Normalize
attention = torch.softmax(attn_scores / (self.head_dim ** (1 / 2)), dim=3)

# attention shape: (N, heads, query_len, key_len)
# values shape: (N, value_len, heads, heads_dim)
# out after matrix multiply: (N, query_len, heads, head_dim), then
# we reshape and concatenate the last two dimensions.
out = torch.einsum("nhql,nlhd->nqhd", [attention, values]).reshape(N, query_len, self.heads * self.head_dim)

print(f"out attn_scores shape: {out.shape}")

out = self.fc_out(out) # (N, query_len, embed_size)
```


Lec 17 | Implementation of Transformer using PyTorch
NPTEL_Transformers.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
MORE VIDEOS
28:16 / 43:33
0s completed at 12:00 PM

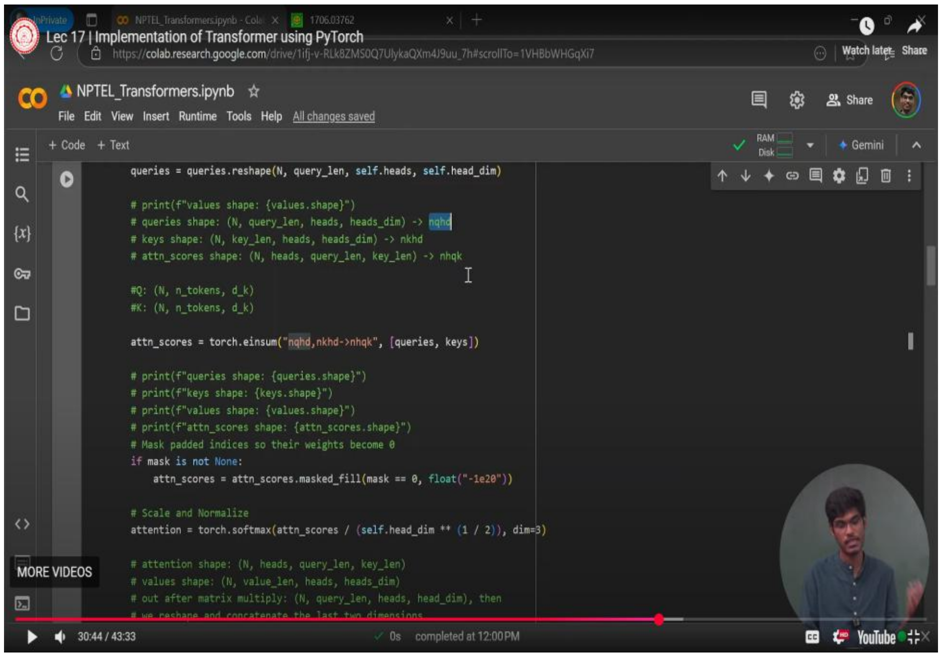
```
# Mask padded indices so their weights become 0
if mask is not None:
    attn_scores = attn_scores.masked_fill(mask == 0, float("-1e20"))

# Scale and Normalize
attention = torch.softmax(attn_scores / (self.head_dim ** (1 / 2)), dim=3)

# attention shape: (N, heads, query_len, key_len)
# values shape: (N, value_len, heads, heads_dim)
# out after matrix multiply: (N, query_len, heads, head_dim), then
# we reshape and concatenate the last two dimensions.
out = torch.einsum("nhql,nlhd->nqhd", [attention, values]).reshape(N, query_len, self.heads * self.head_dim)

print(f"out attn_scores shape: {out.shape}")

out = self.fc_out(out) # (N, query_len, embed_size)

return out
```

```
[43] class TransformerBlock(nn.Module):
         def __init__(self, embed_size, heads, dropout, forward_expansion):
             super(TransformerBlock, self).__init__()
             self.attention = SelfAttention(embed_size, heads)
             self.norm1 = nn.LayerNorm(embed_size)
             self.norm2 = nn.LayerNorm(embed_size)
```



```python
        attention = torch.softmax(attn_scores / (self.head_dim ** (1 / 2)), dim=3)

        # attention shape: (N, heads, query_len, key_len)
        # values shape: (N, value_len, heads, heads_dim)
        # out after matrix multiply: (N, query_len, heads, head_dim), then
        # we reshape and concatenate the last two dimensions.
        out = torch.einsum("nhql,nlhd->nqhd", [attention, values]).reshape(N, query_len, self.heads * self.head_dim)

        print(f"out attn_scores shape: {out.shape}")

        out = self.fc_out(out) # (N, query_len, embed_size)

        return out
```

```python
[43] class TransformerBlock(nn.Module):
    def __init__(self, embed_size, heads, dropout, forward_expansion):
        super(TransformerBlock, self).__init__()
        self.attention = SelfAttention(embed_size, heads)
        self.norm1 = nn.LayerNorm(embed_size)
        self.norm2 = nn.LayerNorm(embed_size)

        self.feed_forward = nn.Sequential(
            nn.Linear(embed_size, forward_expansion * embed_size),
            nn.ReLU(),
            nn.Linear(forward_expansion * embed_size, embed_size),
        )
```



```python
        queries = queries.reshape(N, query_len, self.heads, self.head_dim)

        # print(f"values shape: {values.shape}")
        # queries shape: (N, query_len, heads, heads_dim) -> nqhd
        # keys shape: (N, key_len, heads, heads_dim) -> nkhd
        # attn_scores shape: (N, heads, query_len, key_len) -> nhqk

        #Q: (N, n_tokens, d_k)
        #K: (N, n_tokens, d_k)

        attn_scores = torch.einsum("nqhd,nkhd->nhqk", [queries, keys])

        # print(f"queries shape: {queries.shape}")
        # print(f"keys shape: {keys.shape}")
        # print(f"values shape: {values.shape}")
        # print(f"attn_scores shape: {attn_scores.shape}")
        # Mask padded indices so their weights become 0
        if mask is not None:
            attn_scores = attn_scores.masked_fill(mask == 0, float("-1e20"))

        # Scale and Normalize
        attention = torch.softmax(attn_scores / (self.head_dim ** (1 / 2)), dim=3)

        # attention shape: (N, heads, query_len, key_len)
        # values shape: (N, value_len, heads, heads_dim)
        # out after matrix multiply: (N, query_len, heads, head_dim), then
        # we reshape and concatenate the last two dimensions.
```

So let's go back to our self-attention code. So till now, what we have is our attention score matrix. The attention score matrix is of shape, number of training examples n, number of heads h, and query length cross key length.

So this is clear till now. Now we define a mask. Why do we define a mask? Because suppose the number of the sentences, the different sentences in our training batch are, say, not of equal length. Suppose the first sentence is of length, the number of tokens is, say, 5.

The second sentence is 10. Then we basically pad some spatial tokens into the smaller sentence to make the two sentences of equal length. Sometimes we can also truncate, we can decrease the length of the larger center.

So it's up on us. Generally, we use both truncation and padding. So we truncate the length of the maximum sentence to a particular input length, say like 512. And if any sentence is lesser than 512, we pad it, pad with some special token to make it 512. And if we have sentence greater than 512, we truncate it to 512. So that is how we generally handle batches to make them of same length for parallel processing. So what this mask does is if you have zero in a mask, so we will later when we run this we will see how we define that if we have a padding token make it a zero index right, so if you have a zero what it does is it makes the attention score in place of that mask token as when this mask token is the key for a particular query, the attention score is met basically a large negative number.

So when you have a large negative number think of it as a minus infinity. So when we apply short max it becomes zero so the model does not unnecessarily attend to the mask tokens right So this is kind of a way to handle that the model does not pay attention to the padded tokens, otherwise training will become noise. And then finally, once we have the attention scores, as we saw here, we first scale it by the square root of the head dimension. So that's attention score by self-head square root of that. And we apply softmax to it.

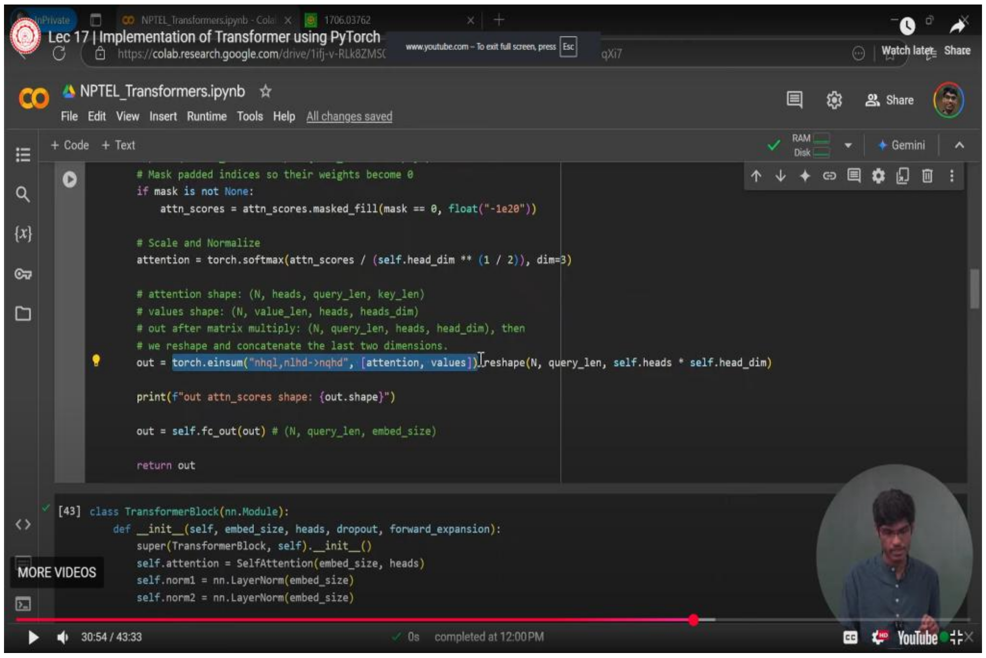
Where do we apply softmax? We apply softmax on dimension 3. So what is dimension 3? So this is 0 dimension, 1 dimension, 2 dimension, 3 dimension. Third dimension is the key length. Applying softmax about dimension 3 tells us that for every query index so what is the for every query index that is for every query token we have a probability distribution right over the keys So that's what attention does. And then again, we use Einsum. So this is again a simple matrix multiplication, but it couldn't have been handled by a simple matrix multiplication.

So what this out is doing is after we have this sort of this, we just need to multiply with the value matrix. So the input is basically the attention and the value matrix. So what is the dimension of the attention? Okay, so attention dimension is the n, h, n is the number of examples, h is the number of heads, q is the query dimension and l is the key dimension.

Basically not key dimension, it is the number of the keys, number of key tokens, key length. And one thing you should always note is the key length is always equal to the value length.

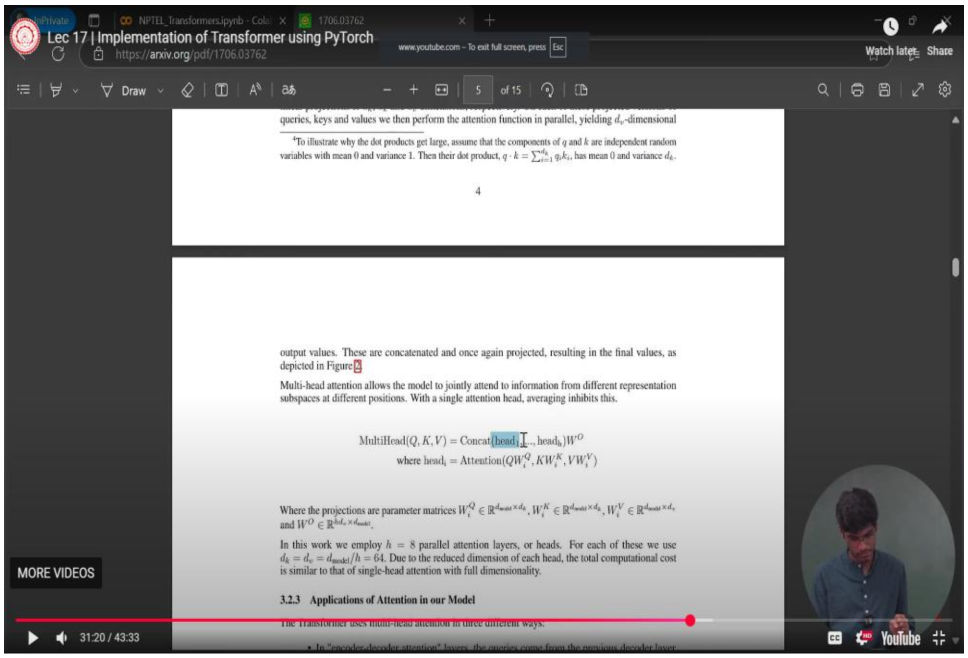
So that key space and the value space are defined over the same set of tokens right? So the key length L and the value length L are the same. It was K previously. We just made it L to make it generalized in both cases. So what is the shape of the value? It's N.

It's value length L, number of its number of dimension. And finally, what we want is for every sentence, we should have NQHD. That is for every sentence, we have the key query length, the head number of heads and number of dimensions that is again we want to restore after multiplying with the value matrix so the final output from each head is same to the final is the dimension of the final output is same as the dimension of the initial input query input we get right.


Lec 17 | Implementation of Transformer using PyTorch
NPTEL_Transformers.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Gemini
Share
MORE VIDEOS
30:54 / 43:33
0s completed at 12:00PM

```
    # Mask padded indices so their weights become 0
    if mask is not None:
        attn_scores = attn_scores.masked_fill(mask == 0, float("-1e20"))

    # Scale and Normalize
    attention = torch.softmax(attn_scores / (self.head_dim ** (1 / 2)), dim=3)

    # attention shape: (N, heads, query_len, key_len)
    # values shape: (N, value_len, heads, heads_dim)
    # out after matrix multiply: (N, query_len, heads, head_dim), then
    # we reshape and concatenate the last two dimensions.
    out = torch.einsum("nhql,nlhd->nqhd", [attention, values]).reshape(N, query_len, self.heads * self.head_dim)

    print(f"out attn_scores shape: {out.shape}")

    out = self.fc_out(out) # (N, query_len, embed_size)

    return out
```

```
[43] class TransformerBlock(nn.Module):
         def __init__(self, embed_size, heads, dropout, forward_expansion):
             super(TransformerBlock, self).__init__()
             self.attention = SelfAttention(embed_size, heads)
             self.norm1 = nn.LayerNorm(embed_size)
             self.norm2 = nn.LayerNorm(embed_size)
```


Lec 17 | Implementation of Transformer using PyTorch
https://arxiv.org/pdf/1706.03762
5 of 15
MORE VIDEOS
31:20 / 43:33

queries, keys and values we then perform the attention function in parallel, yielding d_v-dimensional

[4]To illustrate why the dot products get large, assume that the components of q and k are independent random variables with mean 0 and variance 1. Then their dot product, $q \cdot k = \sum_{i=1}^{d_k} q_i k_i$, has mean 0 and variance d_k.

4

output values. These are concatenated and once again projected, resulting in the final values, as depicted in Figure 2.

Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this.

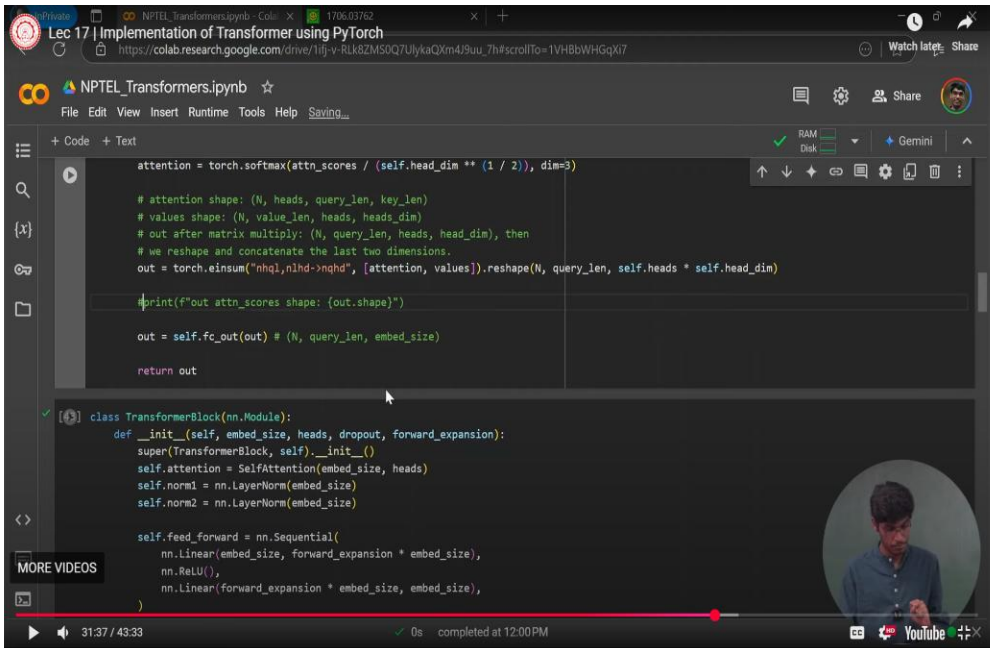
$$\mathrm{MultiHead}(Q,K,V) = \mathrm{Concat}(\mathrm{head}_1, ..., \mathrm{head}_h)W^O$$
$$\text{where } \mathrm{head}_i = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{model}\times d_k}$, $W_i^K \in \mathbb{R}^{d_{model}\times d_k}$, $W_i^V \in \mathbb{R}^{d_{model}\times d_v}$ and $W^O \in \mathbb{R}^{hd_v\times d_{model}}$.

In this work we employ $h = 8$ parallel attention layers, or heads. For each of these we use $d_k = d_v = d_{model}/h = 64$. Due to the reduced dimension of each head, the total computational cost is similar to that of single-head attention with full dimensionality.

3.2.3 Applications of Attention in our Model

The Transformer uses multi-head attention in three different ways:


Lec 17 | Implementation of Transformer using PyTorch
https://colab.research.google.com/drive/1ifj-v-RLk8ZMS0Q7UlykaQXm4J9uu_7h#scrollTo=1VHBbWHGqXi7
NPTEL_Transformers.ipynb
File Edit View Insert Runtime Tools Help Saving...
+ Code + Text
Share
Gemini
MORE VIDEOS
31:37 / 43:33
0s completed at 12:00 PM

```
        attention = torch.softmax(attn_scores / (self.head_dim ** (1 / 2)), dim=3)

        # attention shape: (N, heads, query_len, key_len)
        # values shape: (N, value_len, heads, heads_dim)
        # out after matrix multiply: (N, query_len, heads, head_dim), then
        # we reshape and concatenate the last two dimensions.
        out = torch.einsum("nhql,nlhd->nqhd", [attention, values]).reshape(N, query_len, self.heads * self.head_dim)

        #print(f"out attn_scores shape: {out.shape}")

        out = self.fc_out(out) # (N, query_len, embed_size)

        return out
```

```
class TransformerBlock(nn.Module):
    def __init__(self, embed_size, heads, dropout, forward_expansion):
        super(TransformerBlock, self).__init__()
        self.attention = SelfAttention(embed_size, heads)
        self.norm1 = nn.LayerNorm(embed_size)
        self.norm2 = nn.LayerNorm(embed_size)

        self.feed_forward = nn.Sequential(
            nn.Linear(embed_size, forward_expansion * embed_size),
            nn.ReLU(),
            nn.Linear(forward_expansion * embed_size, embed_size),
        )
```

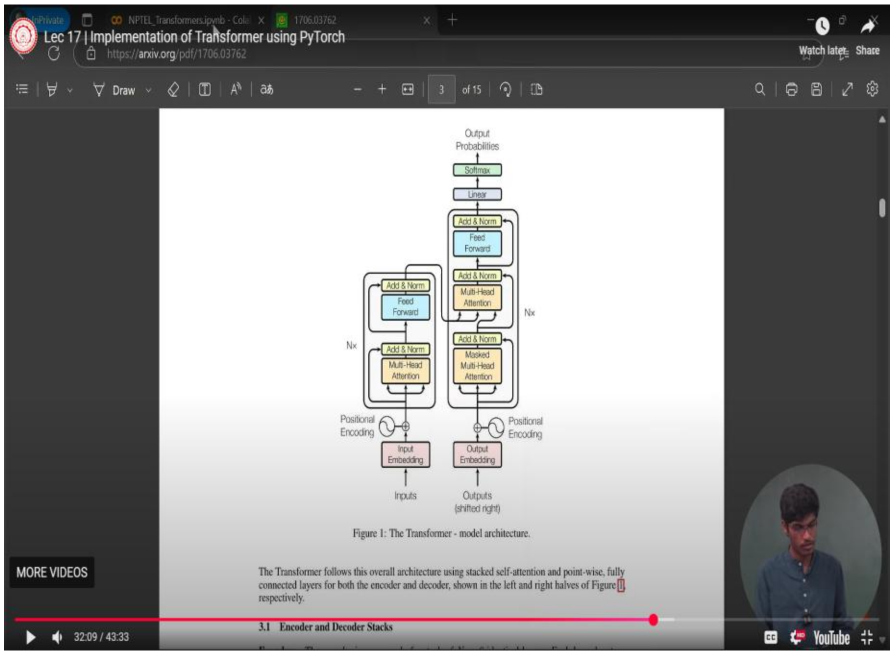


Figure 1: The Transformer - model architecture.

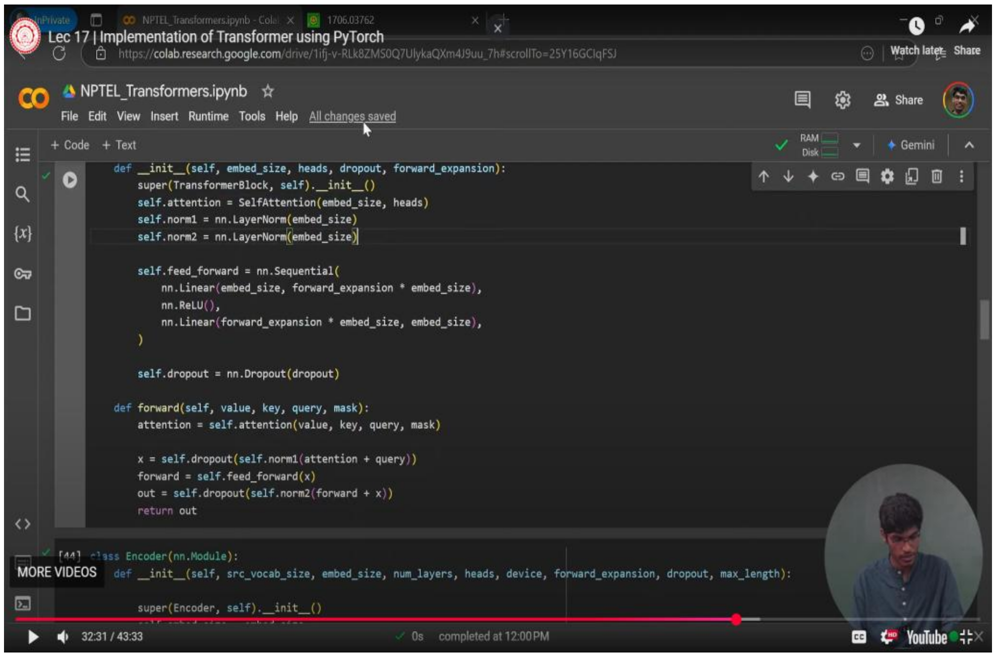
The Transformer follows this overall architecture using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder, shown in the left and right halves of Figure 1, respectively.

3.1 Encoder and Decoder Stacks

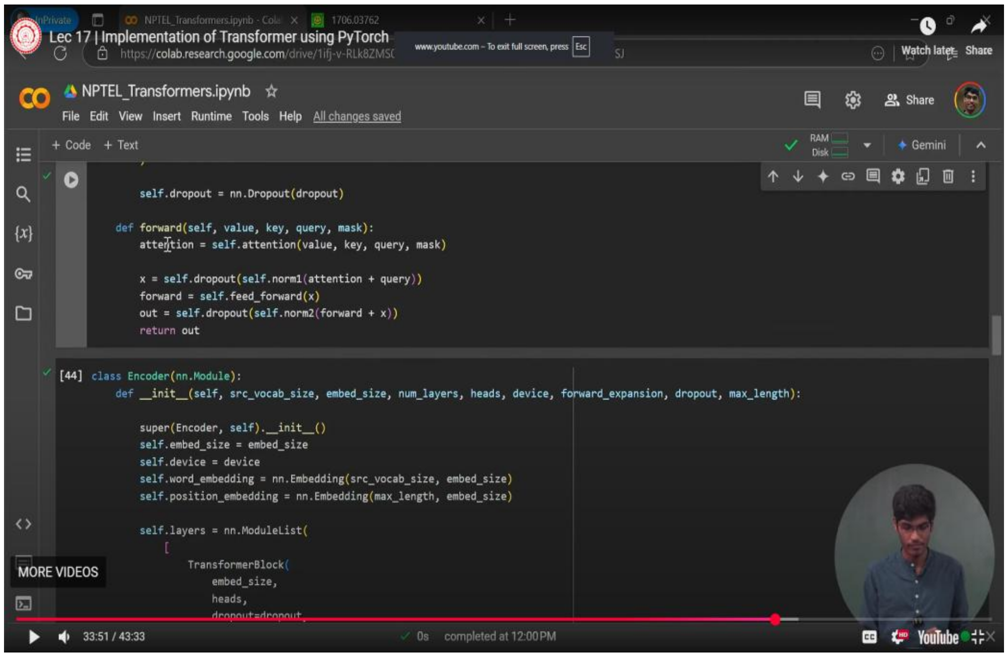
So this makes that and after we have the output so this part what is the output it's the n q h number of heads and d the head dimension. Now we want to reshape it. So why do we want to reshape it? we want to make it basically we want to concatenate all heads right so concatenating all heads means this h and d, so this n and q remains the same, this h and d are merged right.

So basically this operation this concatenation of each head right So this h and d are matched to heads into head dimension, which is the embedding dimension. So finally, you can check. Maybe I can comment this for now. So attention scores dot shape should be the n query length and the embedding dimension.

We simply apply the FC out. FC out is basically applying the WO matrix, right? And we return the out. So now we have this whole multi-head attention module implemented, all over right. We implemented the whole multi-attention. Now the stacking up for encoder and building the decoder is easy, because we have this main part done.


Lec 17 | Implementation of Transformer using PyTorch
NPTEL_Transformers.ipynb
File Edit View Insert Runtime Tools Help All changes saved
MORE VIDEOS
32:31 / 43:33

```
    def __init__(self, embed_size, heads, dropout, forward_expansion):
        super(TransformerBlock, self).__init__()
        self.attention = SelfAttention(embed_size, heads)
        self.norm1 = nn.LayerNorm(embed_size)
        self.norm2 = nn.LayerNorm(embed_size)

        self.feed_forward = nn.Sequential(
            nn.Linear(embed_size, forward_expansion * embed_size),
            nn.ReLU(),
            nn.Linear(forward_expansion * embed_size, embed_size),
        )

        self.dropout = nn.Dropout(dropout)

    def forward(self, value, key, query, mask):
        attention = self.attention(value, key, query, mask)

        x = self.dropout(self.norm1(attention + query))
        forward = self.feed_forward(x)
        out = self.dropout(self.norm2(forward + x))
        return out

[44] class Encoder(nn.Module):
    def __init__(self, src_vocab_size, embed_size, num_layers, heads, device, forward_expansion, dropout, max_length):

        super(Encoder, self).__init__()
```


Lec 17 | Implementation of Transformer using PyTorch
NPTEL_Transformers.ipynb
File Edit View Insert Runtime Tools Help All changes saved
MORE VIDEOS
33:51 / 43:33

```
        self.dropout = nn.Dropout(dropout)

    def forward(self, value, key, query, mask):
        attention = self.attention(value, key, query, mask)

        x = self.dropout(self.norm1(attention + query))
        forward = self.feed_forward(x)
        out = self.dropout(self.norm2(forward + x))
        return out

[44] class Encoder(nn.Module):
    def __init__(self, src_vocab_size, embed_size, num_layers, heads, device, forward_expansion, dropout, max_length):

        super(Encoder, self).__init__()
        self.embed_size = embed_size
        self.device = device
        self.word_embedding = nn.Embedding(src_vocab_size, embed_size)
        self.position_embedding = nn.Embedding(max_length, embed_size)

        self.layers = nn.ModuleList(
            [
                TransformerBlock(
                    embed_size,
                    heads,
                    dropout=dropout,
```

So now we can quickly go through how we finally do the encoder. So for each encoder block we call it transformer block why we will see because this part can be reused while writing the code for the decoder so what do we have we have once we have the multi-attention we take that add not and apply to feed forward network right So we define two norms, norm1 and norm2. So this is norm1 for norm after the multi-attention, and one norm after the feedforward network. We define a simple feedforward network using nn.sequential. If you're not familiar, you can look up the documentation and read what m.sequential does. So, it's simply a linear layer followed by a ReLU activation function and another linear layer.

What this forward expansion does is... So, suppose you have first embedding dimension of D as input. The forward expansion that is the first feed forward layer up projects it. Now, what is the up projection dimension in this paper? So, in this paper, they have taken the forward expansion factor to be 4.

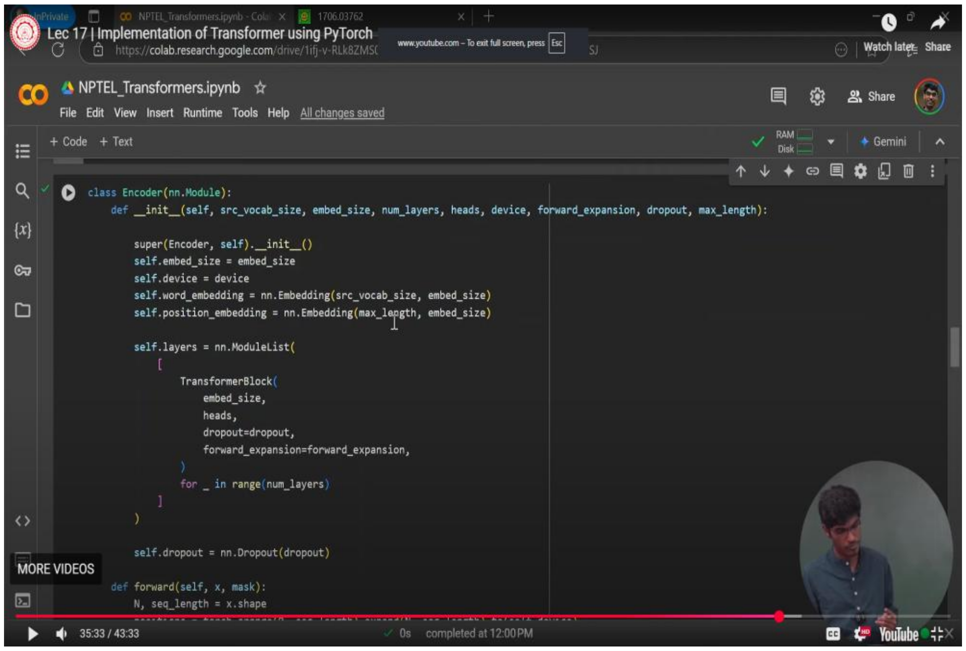
That is... if the input dimension is D, after up-projection, it becomes 4D, right? And then from 4D, it again down-projects to D. So that is what we do in this part, right? We first map it from embedding dimension to forward expansion into embedding dimension, embedding size. So forward expansion can be defined as 4 as in the paper. And then we apply an activation function and then again down-project it from 4D to D, right? And then we apply a dropout.

Dropout is just for regularization, so it doesn't overfit during training. So what does the forward function do? We have the attention. So we have the query key and value matrices and the mask with the padding, which ensures that it doesn't attend to the unnecessary padding tokens, special tokens. So then this attention plus query is basically the add part. So if your attention is the input to the multi-headed self-attention block, so the output is again attention.

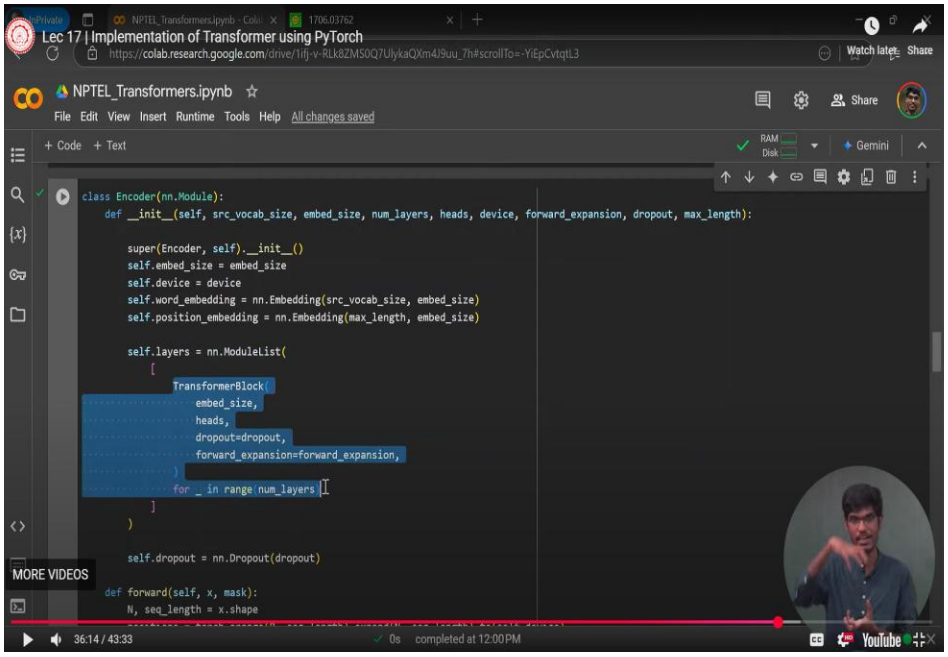
So you can see here, attention. So self.attention, when you pass value query and mask, the output is attention. So with the output of the multi-year attention, you add the initial input query again. This is the add. And then you apply the norm on it. So this X here is basically after multi-attention and addedNorm, right? And then what you do, you pass X to feedForward and again addedNorm.

So what you do, you pass X to feedForward layer. This is forward. And then you again apply the norm. First you add it. So forward is the output of feedForward and X is the input to the feedForward.

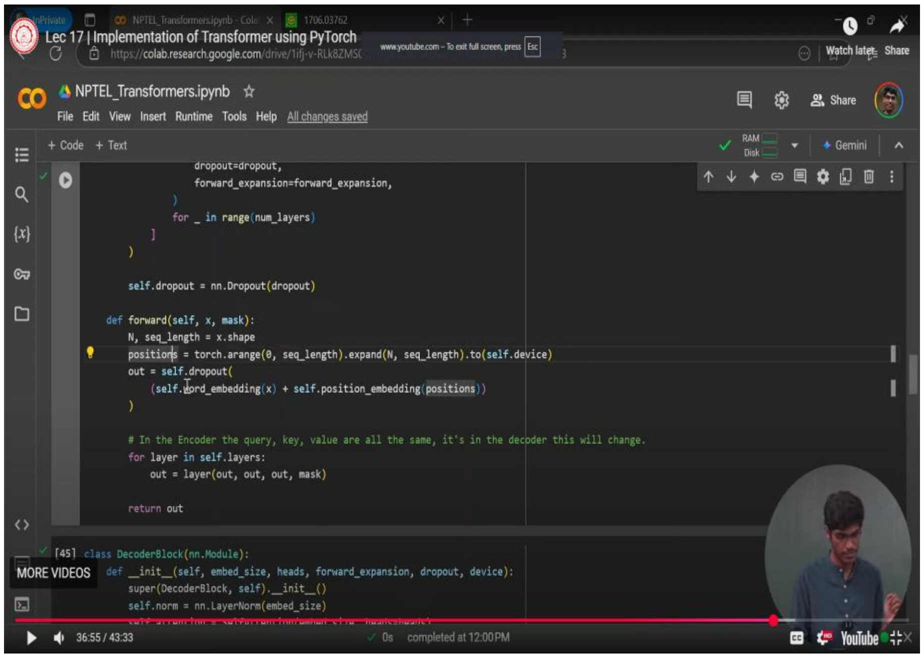
This is the add operation. And then you again apply nilNorm. And finally you get the output. So this completes our encoder block.


Lec 17 | Implementation of Transformer using PyTorch
NPTEL_Transformers.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
MORE VIDEOS
35:33 / 43:33
completed at 12:00 PM

```
class Encoder(nn.Module):
    def __init__(self, src_vocab_size, embed_size, num_layers, heads, device, forward_expansion, dropout, max_length):

        super(Encoder, self).__init__()
        self.embed_size = embed_size
        self.device = device
        self.word_embedding = nn.Embedding(src_vocab_size, embed_size)
        self.position_embedding = nn.Embedding(max_length, embed_size)

        self.layers = nn.ModuleList(
            [
                TransformerBlock(
                    embed_size,
                    heads,
                    dropout=dropout,
                    forward_expansion=forward_expansion,
                )
                for _ in range(num_layers)
            ]
        )

        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask):
        N, seq_length = x.shape
```


Lec 17 | Implementation of Transformer using PyTorch
NPTEL_Transformers.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
MORE VIDEOS
36:14 / 43:33
completed at 12:00 PM

```
class Encoder(nn.Module):
    def __init__(self, src_vocab_size, embed_size, num_layers, heads, device, forward_expansion, dropout, max_length):

        super(Encoder, self).__init__()
        self.embed_size = embed_size
        self.device = device
        self.word_embedding = nn.Embedding(src_vocab_size, embed_size)
        self.position_embedding = nn.Embedding(max_length, embed_size)

        self.layers = nn.ModuleList(
            [
                TransformerBlock(
                    embed_size,
                    heads,
                    dropout=dropout,
                    forward_expansion=forward_expansion,
                )
                for _ in range(num_layers)
            ]
        )

        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask):
        N, seq_length = x.shape
```


Lec 17 | Implementation of Transformer using PyTorch
NPTEL_Transformers.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
MORE VIDEOS
36:55 / 43:33
0s completed at 12:00 PM

```
                    dropout=dropout,
                    forward_expansion=forward_expansion,
                )
                for _ in range(num_layers)
            ]
        )

        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask):
        N, seq_length = x.shape
        positions = torch.arange(0, seq_length).expand(N, seq_length).to(self.device)
        out = self.dropout(
            (self.word_embedding(x) + self.position_embedding(positions))
        )

        # In the Encoder the query, key, value are all the same, it's in the decoder this will change.
        for layer in self.layers:
            out = layer(out, out, out, mask)

        return out
```

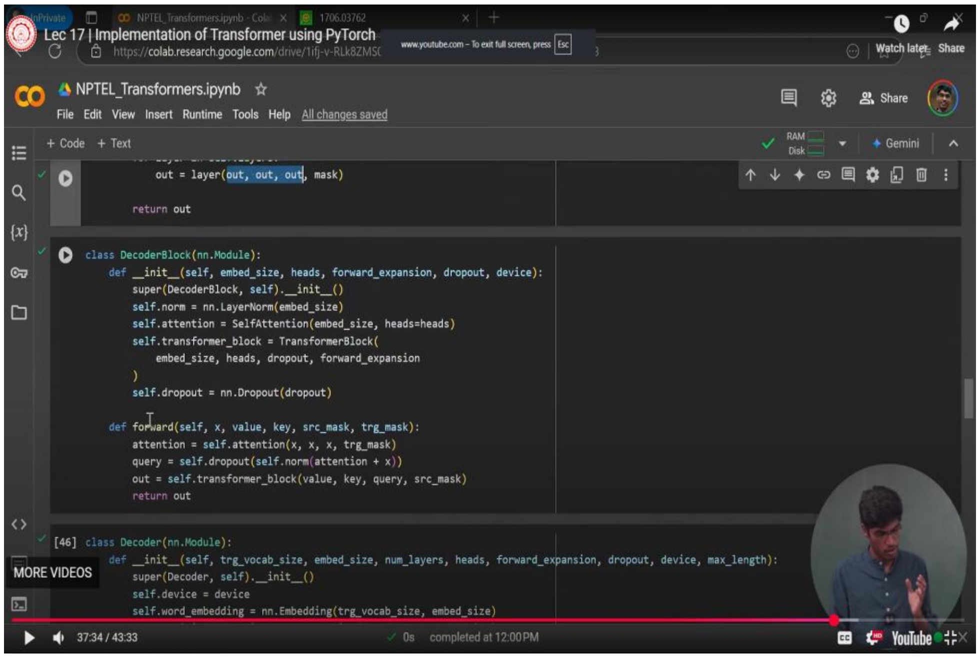
```
[45] class DecoderBlock(nn.Module):
    def __init__(self, embed_size, heads, forward_expansion, dropout, device):
        super(DecoderBlock, self).__init__()
        self.norm = nn.LayerNorm(embed_size)
```

Now, for the whole encoder module, what you do? You basically stack n encoder blocks, one on top of the other. And what is the input? The input is you have the word embedding, you add positional embedding to it. We have taken a more similar simpler trainable positional embedding layer here which is trainable.

So the embedding is defined over the max length of the input and the embedding dimension right. And the word embedding is embedding add-on embedding is basically a trainable dictionary lookup table kind of thing right. So the number of keys are the vocabulary size and the embedding dimension, so if we can quickly go through this that we have a module list right so in the module list what it does is it stacks in modules right. So this transformer block is stacked in types. So what does that mean? Is the input? You give input to one block, and then the output of the block is produced acts as the input to the block on top of it. So we stack that. We add a dropout. So if you look at the forward function, first you have the x, which is the input. The x is basically a matrix. The number of rows is the number of training examples in the batch.

And the sequence length is the number of tokens in each. in each sentence. So positions are basically from 0 to the word length. These are used for positional embeddings. So what you do first is get the embeddings of each token.

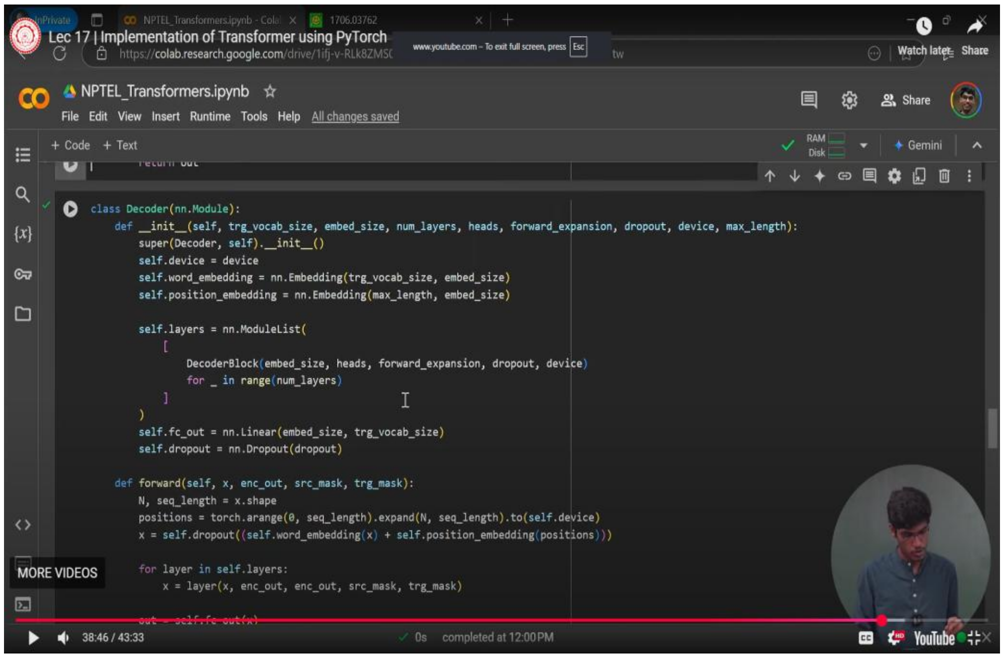
And add it to the positional embedding. And then for each layer in self.layers, that means for each encoder block, You passed on the final out. So what is out? Out is x plus the positional embedding of x. Right, so you pass on that, so the key query and value. As I said, the key query and value for the encoder block are the same. It's the input embedding plus the position encoding embedding that matrix, right? So the key query value comes from the same matrix in the case of the encoder.


Lec 17 | Implementation of Transformer using PyTorch
NPTEL_Transformers.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
MORE VIDEOS
37:34 / 43:33
completed at 12:00 PM

```
            out = layer(out, out, out, mask)

        return out
```

```
class DecoderBlock(nn.Module):
    def __init__(self, embed_size, heads, forward_expansion, dropout, device):
        super(DecoderBlock, self).__init__()
        self.norm = nn.LayerNorm(embed_size)
        self.attention = SelfAttention(embed_size, heads=heads)
        self.transformer_block = TransformerBlock(
            embed_size, heads, dropout, forward_expansion
        )
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, value, key, src_mask, trg_mask):
        attention = self.attention(x, x, x, trg_mask)
        query = self.dropout(self.norm(attention + x))
        out = self.transformer_block(value, key, query, src_mask)
        return out
```

```
[46] class Decoder(nn.Module):
    def __init__(self, trg_vocab_size, embed_size, num_layers, heads, forward_expansion, dropout, device, max_length):
        super(Decoder, self).__init__()
        self.device = device
        self.word_embedding = nn.Embedding(trg_vocab_size, embed_size)
```


Lec 17 | Implementation of Transformer using PyTorch
NPTEL_Transformers.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
MORE VIDEOS
38:46 / 43:33
completed at 12:00 PM

```
class Decoder(nn.Module):
    def __init__(self, trg_vocab_size, embed_size, num_layers, heads, forward_expansion, dropout, device, max_length):
        super(Decoder, self).__init__()
        self.device = device
        self.word_embedding = nn.Embedding(trg_vocab_size, embed_size)
        self.position_embedding = nn.Embedding(max_length, embed_size)

        self.layers = nn.ModuleList(
            [
                DecoderBlock(embed_size, heads, forward_expansion, dropout, device)
                for _ in range(num_layers)
            ]
        )
        self.fc_out = nn.Linear(embed_size, trg_vocab_size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, enc_out, src_mask, trg_mask):
        N, seq_length = x.shape
        positions = torch.arange(0, seq_length).expand(N, seq_length).to(self.device)
        x = self.dropout((self.word_embedding(x) + self.position_embedding(positions)))

        for layer in self.layers:
            x = layer(x, enc_out, enc_out, src_mask, trg_mask)
```

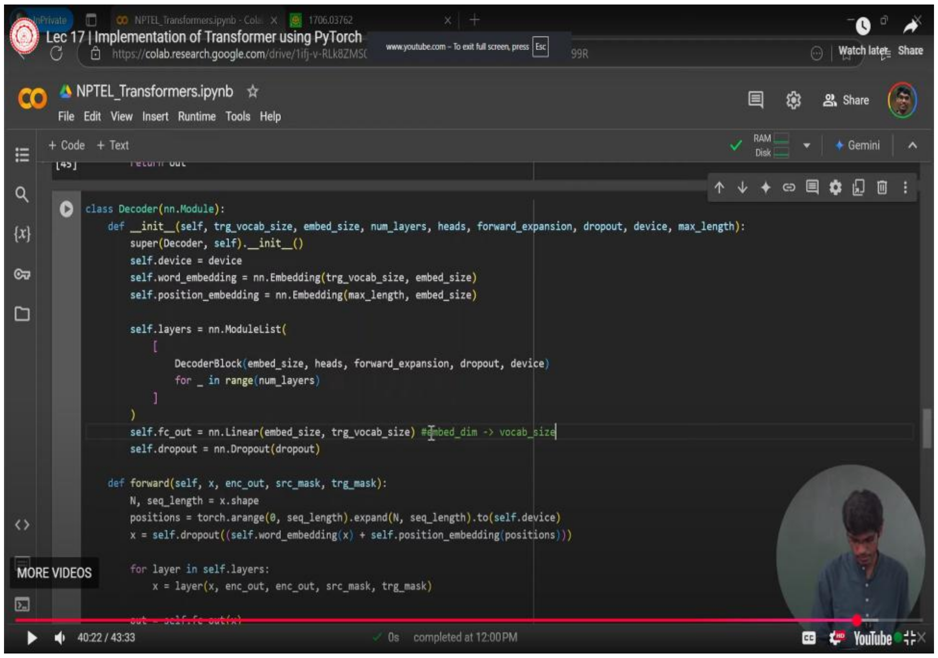
On the other hand, for the decoder block on the right, So for the decoder block, everything is the same; the only difference is that The key query and value of the query come from the input. and the key and value come from the encoders. The first attention which is the masked multi attention.

This masking is done on the future tokens, if you remember. So the Transformers can't see the future. We should ensure the transformer shouldn't see the future Because generally, the transformer can see the future. If you give the whole target sentence as input. So we must make sure that the future tokens are masked as keys and values when we are handling a particular token as a query. So they shouldn't be able to see the future. So that is the first thing; that is the masking. And this masked multi-attention takes as input the input to the decoder. So the first attention block takes as input the input to the decoder. And then we apply that norm again, similar to that.

And then we apply this in the cross attention, right? Cross attention is the key. The value comes from the encoder, and the query comes from the decoder input. So basically, when we stack these layers in the decoder again, We have the word embedding and positional embedding; we stack the N layers, right? And this is the feedforward, basically FC out. similar to what we did in encoder, and we add a dropout. So this is the input to the decoder, and the positions.

So what we do is we first take the word embedding and add the position embedding. Add a dropout, and then for every layer, what we do is basically. We just pass on the current input to the decoder, the output of the encoder. and the source mask for masking the padded tokens and a target mask for masking the padded tokens Target mask is basically for masking the tokens. As I said in the decoder, what is the mask? The future tokens, right? So target mask marks the future tokens.

So the transformer can't see the future in that case. And this output is basically projecting whatever output you have. So this fc_out is this final linear layer, right? It projects from the embedding dimension to the vocabulary dimensions. And then softmax makes it a probability distribution, right?

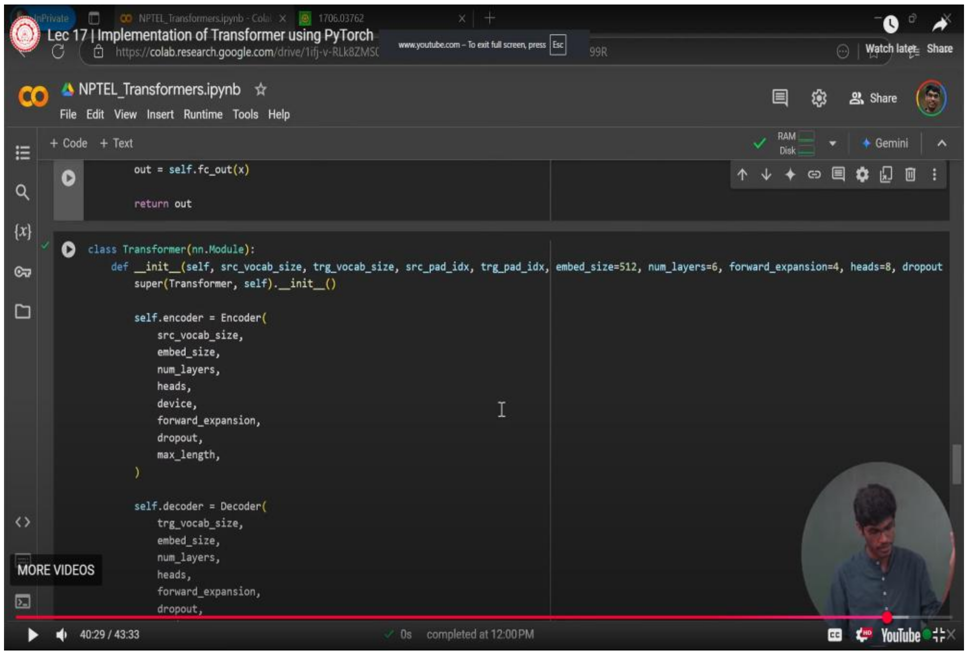

Lec 17 | Implementation of Transformer using PyTorch

NPTEL_Transformers.ipynb

```python
        return out
```

```python
class Decoder(nn.Module):
    def __init__(self, trg_vocab_size, embed_size, num_layers, heads, forward_expansion, dropout, device, max_length):
        super(Decoder, self).__init__()
        self.device = device
        self.word_embedding = nn.Embedding(trg_vocab_size, embed_size)
        self.position_embedding = nn.Embedding(max_length, embed_size)

        self.layers = nn.ModuleList(
            [
                DecoderBlock(embed_size, heads, forward_expansion, dropout, device)
                for _ in range(num_layers)
            ]
        )
        self.fc_out = nn.Linear(embed_size, trg_vocab_size) #embed_dim -> vocab_size
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, enc_out, src_mask, trg_mask):
        N, seq_length = x.shape
        positions = torch.arange(0, seq_length).expand(N, seq_length).to(self.device)
        x = self.dropout((self.word_embedding(x) + self.position_embedding(positions)))

        for layer in self.layers:
            x = layer(x, enc_out, enc_out, src_mask, trg_mask)

        out = self.fc_out(x)
```

40:22 / 43:33 ✓ 0s completed at 12:00 PM

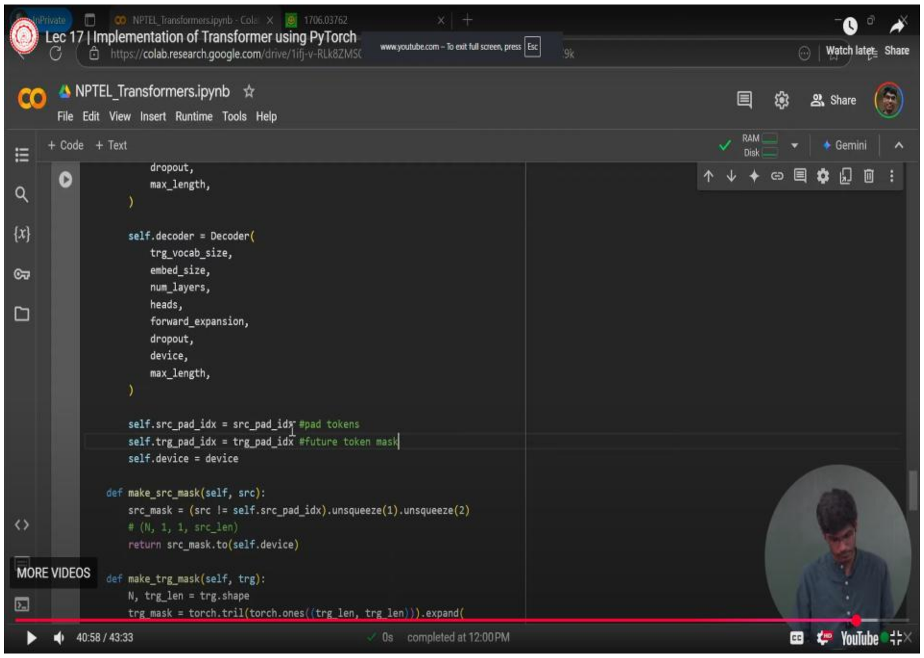

Lec 17 | Implementation of Transformer using PyTorch

NPTEL_Transformers.ipynb

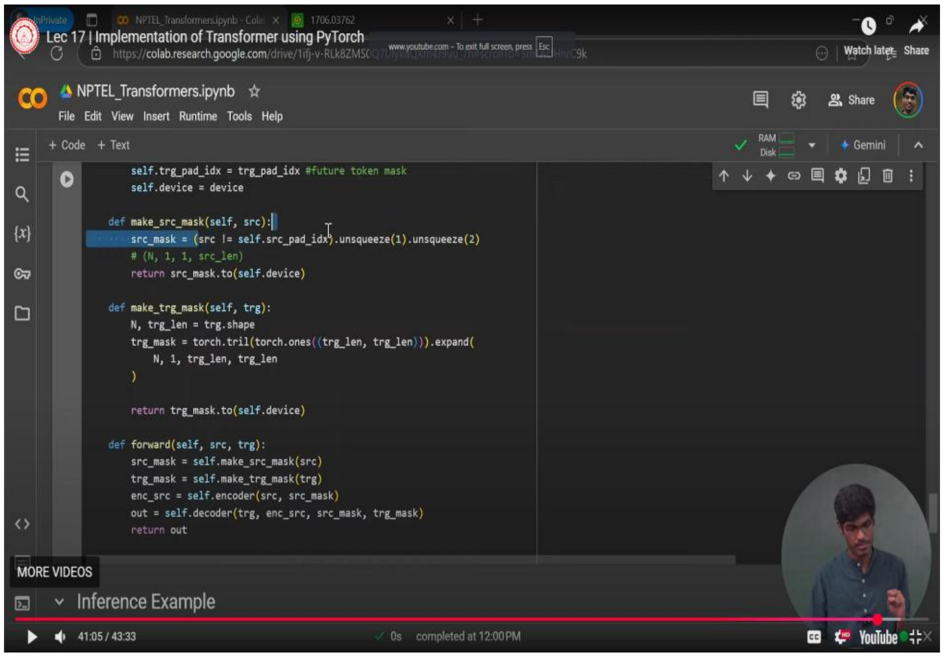
```python
        out = self.fc_out(x)

        return out
```

```python
class Transformer(nn.Module):
    def __init__(self, src_vocab_size, trg_vocab_size, src_pad_idx, trg_pad_idx, embed_size=512, num_layers=6, forward_expansion=4, heads=8, dropout
        super(Transformer, self).__init__()

        self.encoder = Encoder(
            src_vocab_size,
            embed_size,
            num_layers,
            heads,
            device,
            forward_expansion,
            dropout,
            max_length,
        )

        self.decoder = Decoder(
            trg_vocab_size,
            embed_size,
            num_layers,
            heads,
            forward_expansion,
            dropout,
```

40:29 / 43:33 ✓ 0s completed at 12:00 PM

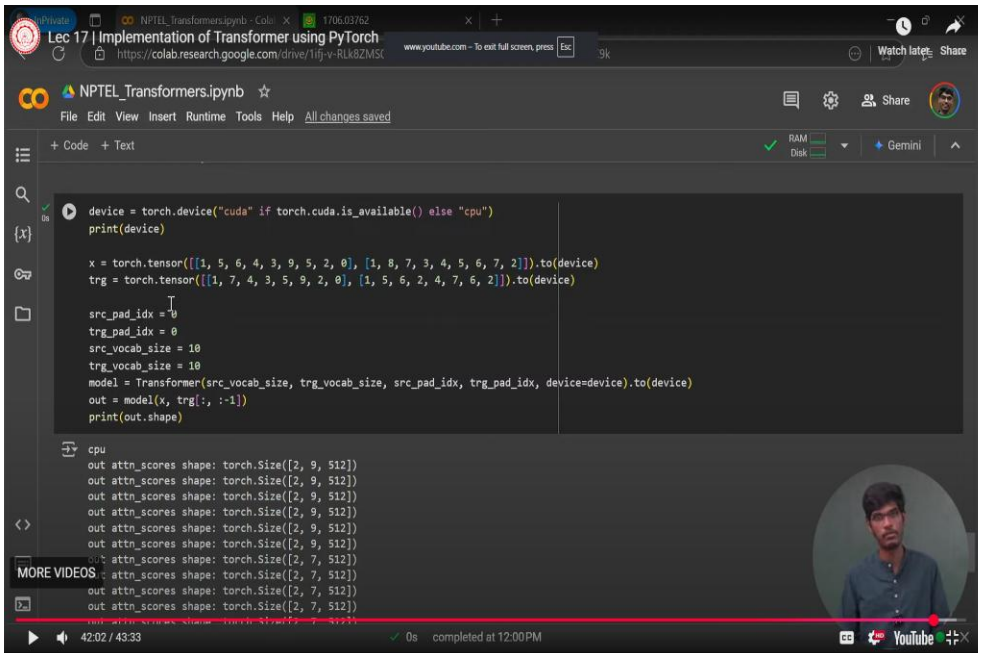

Lec 17 | Implementation of Transformer using PyTorch
NPTEL_Transformers.ipynb
File Edit View Insert Runtime Tools Help
+ Code + Text
MORE VIDEOS
40:58 / 43:33
0s completed at 12:00 PM

```
            dropout,
            max_length,
        )

        self.decoder = Decoder(
            trg_vocab_size,
            embed_size,
            num_layers,
            heads,
            forward_expansion,
            dropout,
            device,
            max_length,
        )

        self.src_pad_idx = src_pad_idx #pad tokens
        self.trg_pad_idx = trg_pad_idx #future token mask
        self.device = device

    def make_src_mask(self, src):
        src_mask = (src != self.src_pad_idx).unsqueeze(1).unsqueeze(2)
        # (N, 1, 1, src_len)
        return src_mask.to(self.device)

    def make_trg_mask(self, trg):
        N, trg_len = trg.shape
        trg_mask = torch.tril(torch.ones((trg_len, trg_len))).expand(
```


Lec 17 | Implementation of Transformer using PyTorch
NPTEL_Transformers.ipynb
File Edit View Insert Runtime Tools Help
+ Code + Text
MORE VIDEOS
41:05 / 43:33
0s completed at 12:00 PM

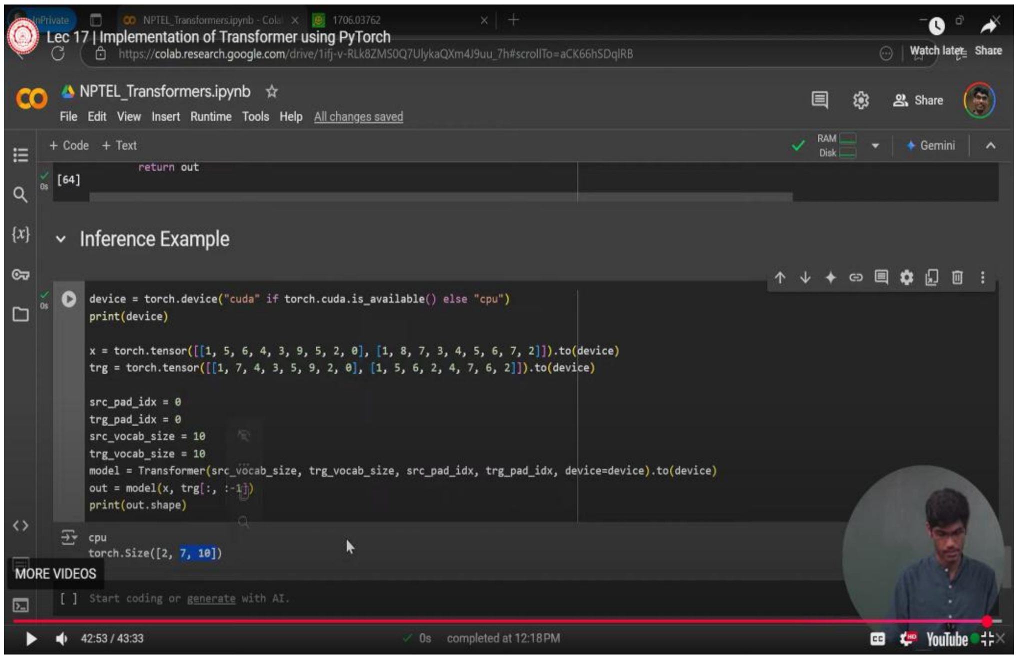
```
        self.trg_pad_idx = trg_pad_idx #future token mask
        self.device = device

    def make_src_mask(self, src):
        src_mask = (src != self.src_pad_idx).unsqueeze(1).unsqueeze(2)
        # (N, 1, 1, src_len)
        return src_mask.to(self.device)

    def make_trg_mask(self, trg):
        N, trg_len = trg.shape
        trg_mask = torch.tril(torch.ones((trg_len, trg_len))).expand(
            N, 1, trg_len, trg_len
        )

        return trg_mask.to(self.device)

    def forward(self, src, trg):
        src_mask = self.make_src_mask(src)
        trg_mask = self.make_trg_mask(trg)
        enc_src = self.encoder(src, src_mask)
        out = self.decoder(trg, enc_src, src_mask, trg_mask)
        return out
```

Inference Example


Lec 17 | Implementation of Transformer using PyTorch
NPTEL_Transformers.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
MORE VIDEOS
42:02 / 43:33
completed at 12:00PM

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)

x = torch.tensor([[1, 5, 6, 4, 3, 9, 5, 2, 0], [1, 8, 7, 3, 4, 5, 6, 7, 2]]).to(device)
trg = torch.tensor([[1, 7, 4, 3, 5, 9, 2, 0], [1, 5, 6, 2, 4, 7, 6, 2]]).to(device)

src_pad_idx = 0
trg_pad_idx = 0
src_vocab_size = 10
trg_vocab_size = 10
model = Transformer(src_vocab_size, trg_vocab_size, src_pad_idx, trg_pad_idx, device=device).to(device)
out = model(x, trg[:, :-1])
print(out.shape)
```

```
cpu
out attn_scores shape: torch.Size([2, 9, 512])
out attn_scores shape: torch.Size([2, 9, 512])
out attn_scores shape: torch.Size([2, 9, 512])
out attn_scores shape: torch.Size([2, 9, 512])
out attn_scores shape: torch.Size([2, 9, 512])
out attn_scores shape: torch.Size([2, 9, 512])
out attn_scores shape: torch.Size([2, 7, 512])
out attn_scores shape: torch.Size([2, 7, 512])
out attn_scores shape: torch.Size([2, 7, 512])
out attn_scores shape: torch.Size([2, 7, 512])
```


Lec 17 | Implementation of Transformer using PyTorch
NPTEL_Transformers.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
MORE VIDEOS
42:53 / 43:33
completed at 12:18PM

```
        return out
```

[64]

Inference Example

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)

x = torch.tensor([[1, 5, 6, 4, 3, 9, 5, 2, 0], [1, 8, 7, 3, 4, 5, 6, 7, 2]]).to(device)
trg = torch.tensor([[1, 7, 4, 3, 5, 9, 2, 0], [1, 5, 6, 2, 4, 7, 6, 2]]).to(device)

src_pad_idx = 0
trg_pad_idx = 0
src_vocab_size = 10
trg_vocab_size = 10
model = Transformer(src_vocab_size, trg_vocab_size, src_pad_idx, trg_pad_idx, device=device).to(device)
out = model(x, trg[:, :-1])
print(out.shape)
```

```
cpu
torch.Size([2, 7, 10])
```

```
[ ] Start coding or generate with AI.
```

So this output is basically projecting from embedding dimension to the vocab size, right? And this completes the decoder. And the final part is using all of this.

So, when we use all this, what happens? So we have an encoder. We define, just as we said, the other n encoder models within it. We define a decoder. We define the padding index, right? Padding index for input to the encoder; padding index for input to the decoder.

So this takes care of the bad tokens. and this takes care of the future token right. and then we have device. So then we have the source mask, so what does the source do? For all places where there is zero, we'll define the pad index to be zero. It's a source mark, so it's one everywhere, and wherever there is a padding token, it's zero. And previously, we saw how to make it minus infinity in the attention matrix. Similarly, for the target mask, the trill is basically a lower triangular matrix.

So, as we have seen in the lecture, for the target mask, What do we have? It's a lower triangular matrix. So, the lower triangular matrix is 1. And the upper triangular matrix is basically negative infinity in the attention score.

So we make it a 0 here. And then in the mask, whatever is 0 becomes minus infinity. As we discussed in the code. And then simply pass the forward function, the source mask, and the target mask. We get the output from the encoder using the CellBread encoder. and we pass the output of the encoder to the decoder along with the input to the decoder.

So, just to show, suppose we have basically a tokenized input. So this is, say, a tokenized input of dimensions 2 and 9. And then we define the source and target pad indices to be 0. and the woke-up size is to be 10. And then if we input this x, input X, the target book of size, sole book of size and all this to the transformer, it will process it. So maybe I will run this once to show you that it works well, right? So this print that is working on the CPU, there's a printing device over here and then the torch dot size, which is two, is the number of example sentences. Seven is the number of output tokens, and 10 is... Basically, the output vocabulary size. So this is a kind of walkthrough on how we can implement transformers in PyTorch. We could do more things, like defining the position encoding using rope.

And things like that. But this must be helpful for you to at least get started. With coding transformers in PyTorch. So that's it for this lecture. Thank you. Enjoy and happy learning! You.