Department of Computer Science and Engineering (Data Science)

Advanced Computational Linguistics
Experiment No 6

Name: Umang Kirit Lodaya **SAP ID: 60009200032**

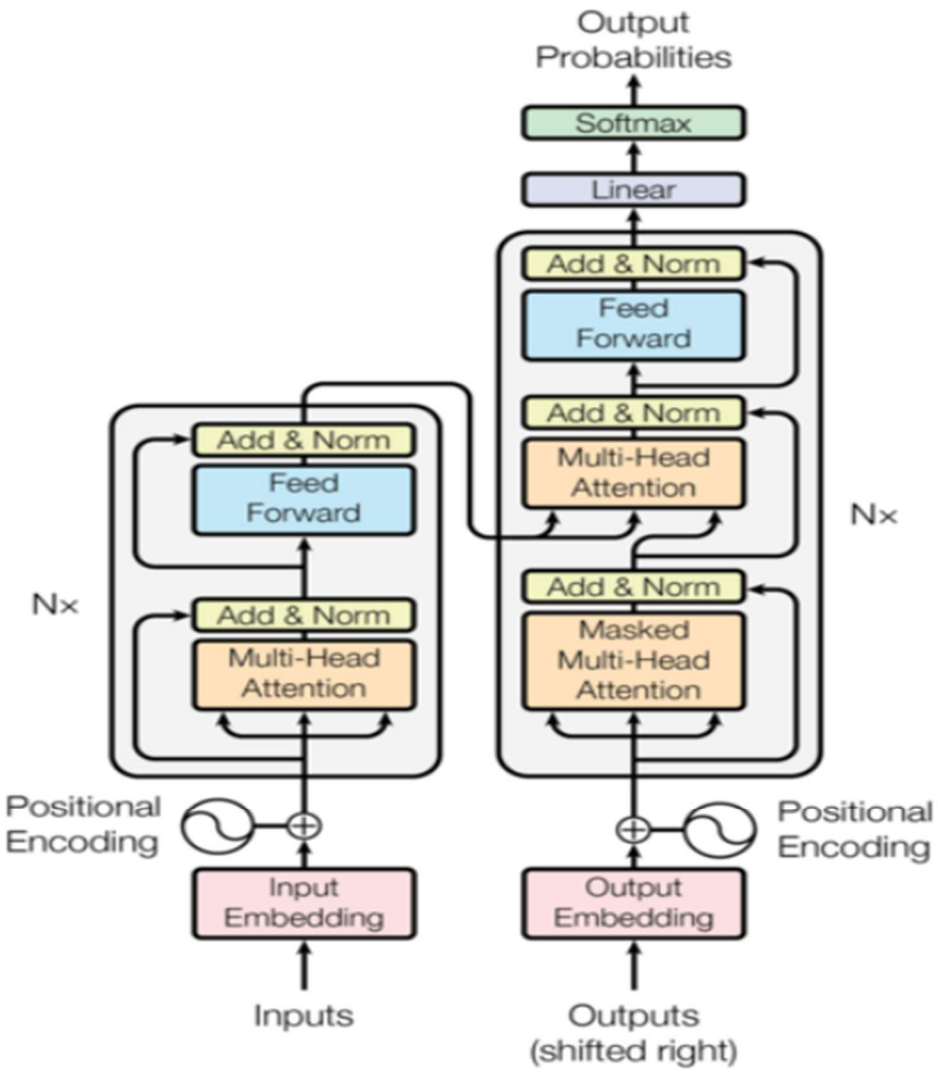
Batch: D11

Aim: Implementation of Transformer model using PyTorch

Theory:

Introduction

Transformers have an encoder-decoder architecture. It is common for language translation models. The Transformer model, introduced by Vaswani et al. in the paper "Attention is All You Need," is a deep learning architecture designed for sequence-to-sequence tasks, such as machine translation and text summarization. It is based on self-attention mechanisms and has become the foundation for many state-of-the-art natural language processing models, like GPT and BERT.

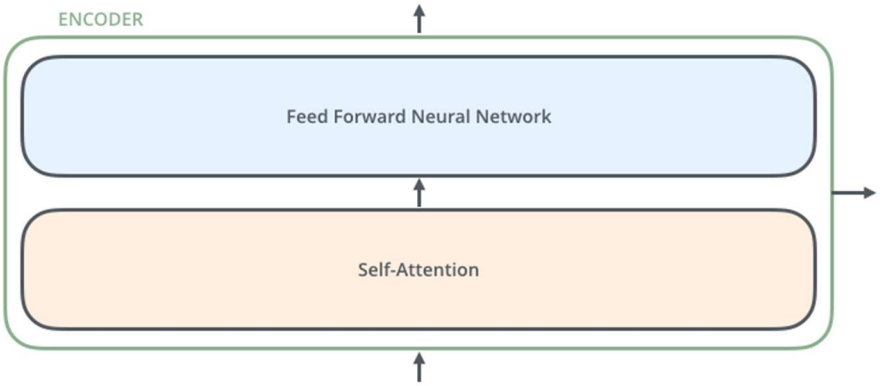


Encoder and Decoder

The transformer uses an encoder-decoder architecture. The encoder (left) processes the input sequence and returns a *feature vector* (or *memory* vector). The encoding component is a stack of encoders (the paper stacks six of them on top of each other – there's nothing magical about the number six, one can experiment with other arrangements). The decoding component is a stack of decoders of the same number. The decoder processes the target sequence and incorporates information from the encoder memory. The output from the decoder is our model's prediction!

We can code the encoder/decoder modules independently of one another, and then combine them at the end.

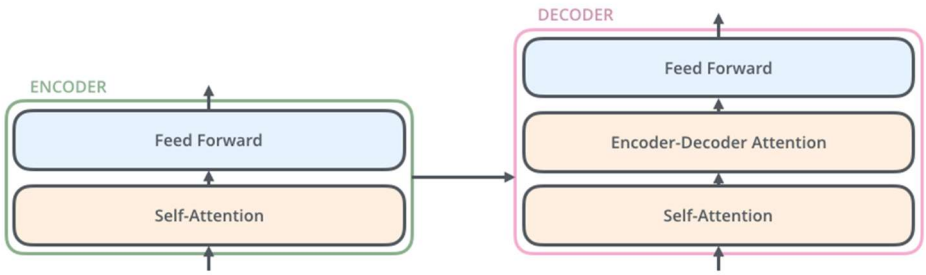
The encoders are all identical in structure (yet they do not share weights). Each one is broken down into two sub-layers:



The encoder's inputs first flow through a self-attention layer – a layer that helps the encoder look at other words in the input sentence as it encodes a specific word. We'll look closer at self-attention later in the post.

The outputs of the self-attention layer are fed to a feed-forward neural network. The exact same feed-forward network is independently applied to each position.

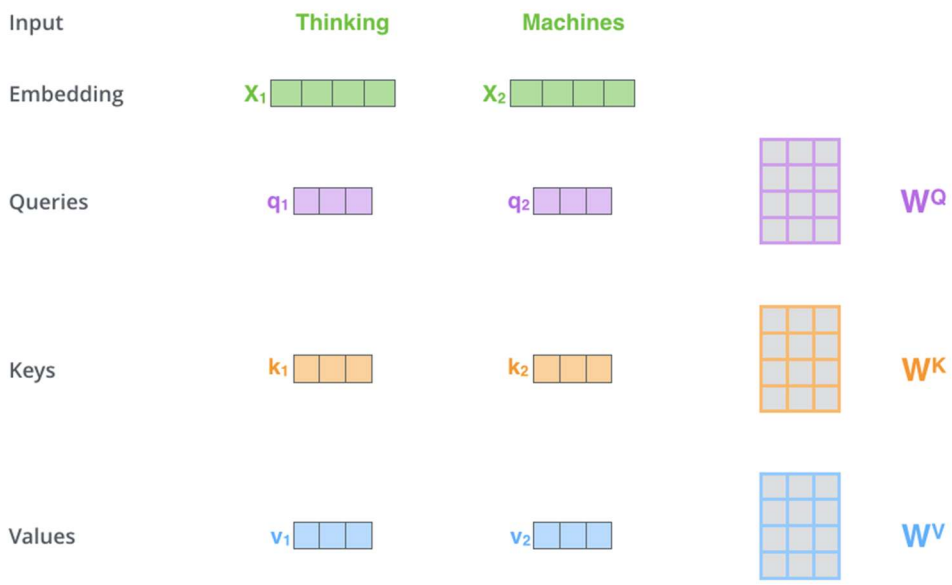
The decoder has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence (similar what attention does in seq2seq models).



Self-Attention

The first step in calculating self-attention is to create three vectors from each of the encoder's input vectors (in this case, the embedding of each word). So for each word, we create a Query vector, a Key vector, and a Value vector. These vectors are created by multiplying the embedding by three matrices that we trained during the training process.

Notice that these new vectors are smaller in dimension than the embedding vector. Their dimensionality is 64, while the embedding and encoder input/output vectors have dimensionality of 512. They don't HAVE to be smaller, this is an architecture choice to make the computation of multiheaded attention (mostly) constant.



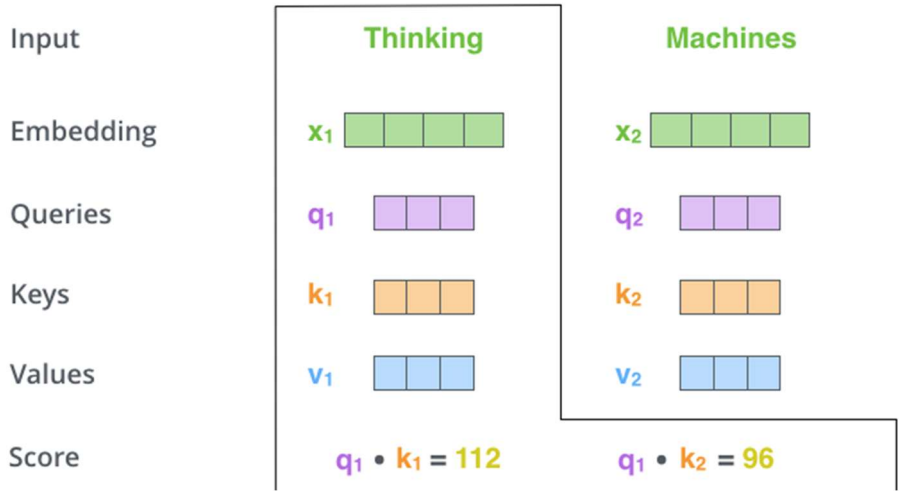
Multiplying x1 by the WQ weight matrix produces q1, the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.

What are the "query", "key", and "value" vectors?

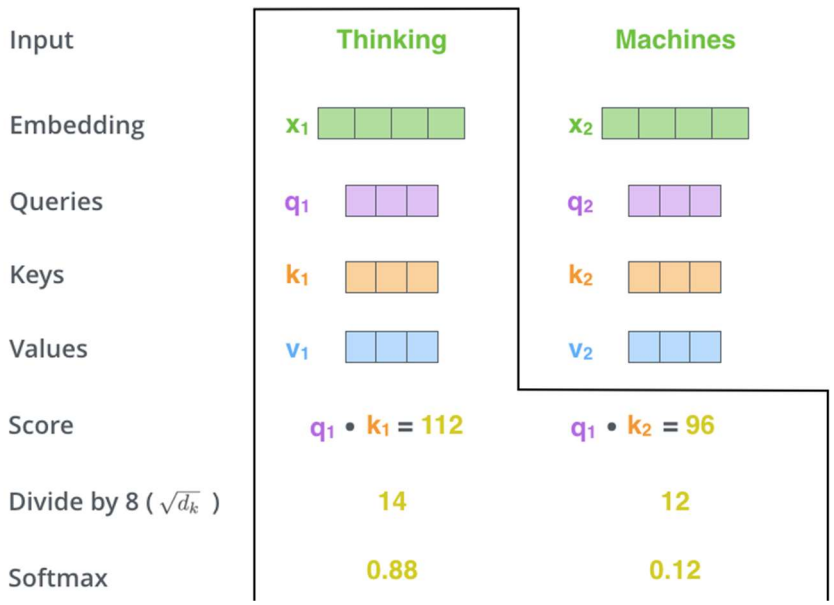
They're abstractions that are useful for calculating and thinking about attention. Once you proceed with reading how attention is calculated below, you'll know pretty much all you need to know about the role each of these vectors plays.

The second step in calculating self-attention is to calculate a score. Say we're calculating the self-attention for the first word in this example, "Thinking". We need to score each word of the input sentence against this word. The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.

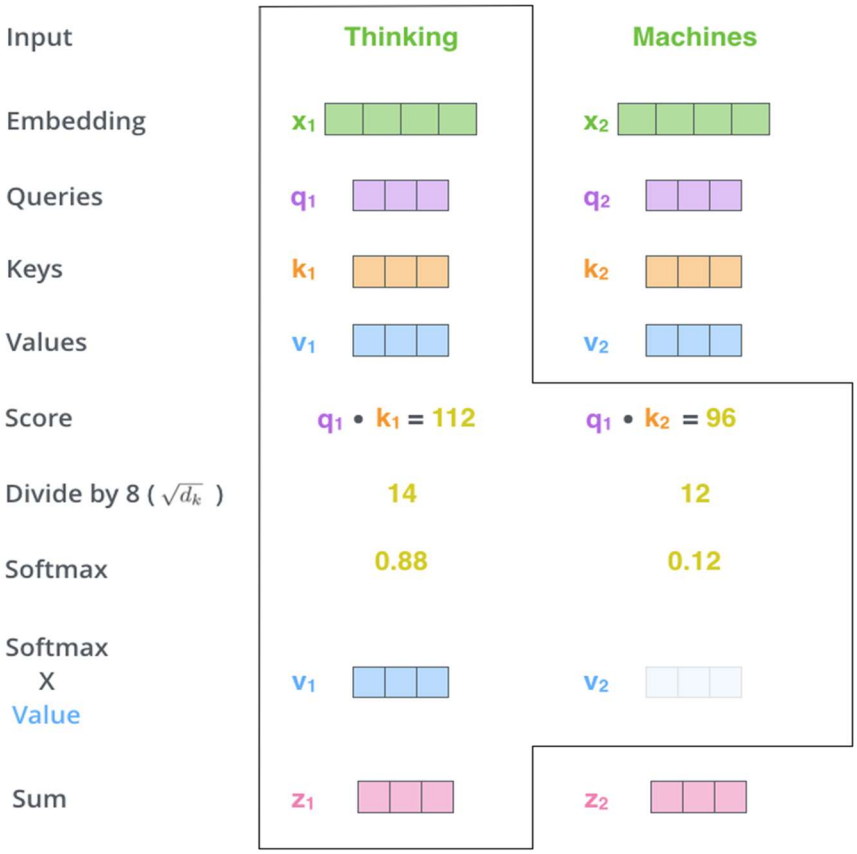
The score is calculated by taking the dot product of the query vector with the key vector of the respective word we're scoring. So if we're processing the self-attention for the word in position #1, the first score would be the dot product of q1 and k1. The second score would be the dot product of q1 and k2.



The third and fourth steps are to divide the scores by 8 (the square root of the dimension of the key vectors used in the paper – 64. This leads to having more stable gradients. There could be other possible values here, but this is the default), then pass the result through a softmax operation. Softmax normalizes the scores so they're all positive and add up to 1.



This softmax score determines how much each word will be expressed at this position. Clearly the word at this position will have the highest softmax score, but sometimes it's useful to attend to another word that is relevant to the current word.



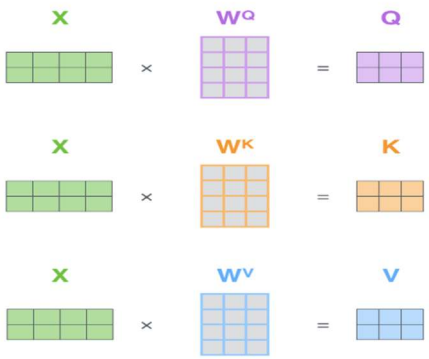
The fifth step is to multiply each value vector by the softmax score (in preparation to sum them up). The intuition here is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words (by multiplying them by tiny numbers like 0.001, for example).

The sixth step is to sum up the weighted value vectors. This produces the output of the self-attention layer at this position (for the first word).

That concludes the self-attention calculation. The resulting vector is one we can send along to the feed-forward neural network. In the actual implementation, however, this calculation is done in matrix form for faster processing. So let's look at that now that we've seen the intuition of the calculation on the word level.

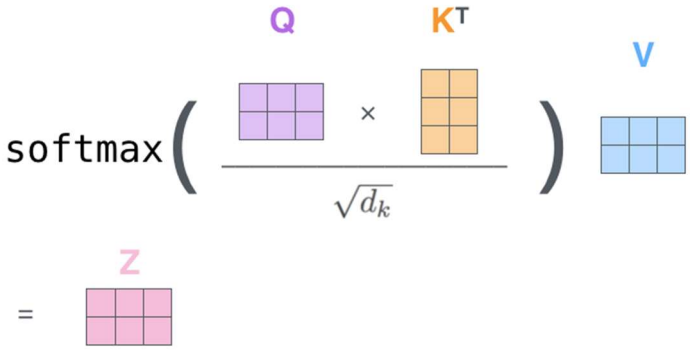
Matrix Calculation of Self-Attention

The first step is to calculate the Query, Key, and Value matrices. We do that by packing our embeddings into a matrix X, and multiplying it by the weight matrices we've trained (WQ, WK, WV).



Every row in the X matrix corresponds to a word in the input sentence. We again see the difference in size of the embedding vector (512, or 4 boxes in the figure), and the q/k/v vectors (64, or 3 boxes in the figure)

Finally, since we're dealing with matrices, we can condense steps two through six in one formula to calculate the outputs of the self-attention layer.



The self-attention calculation in matrix form

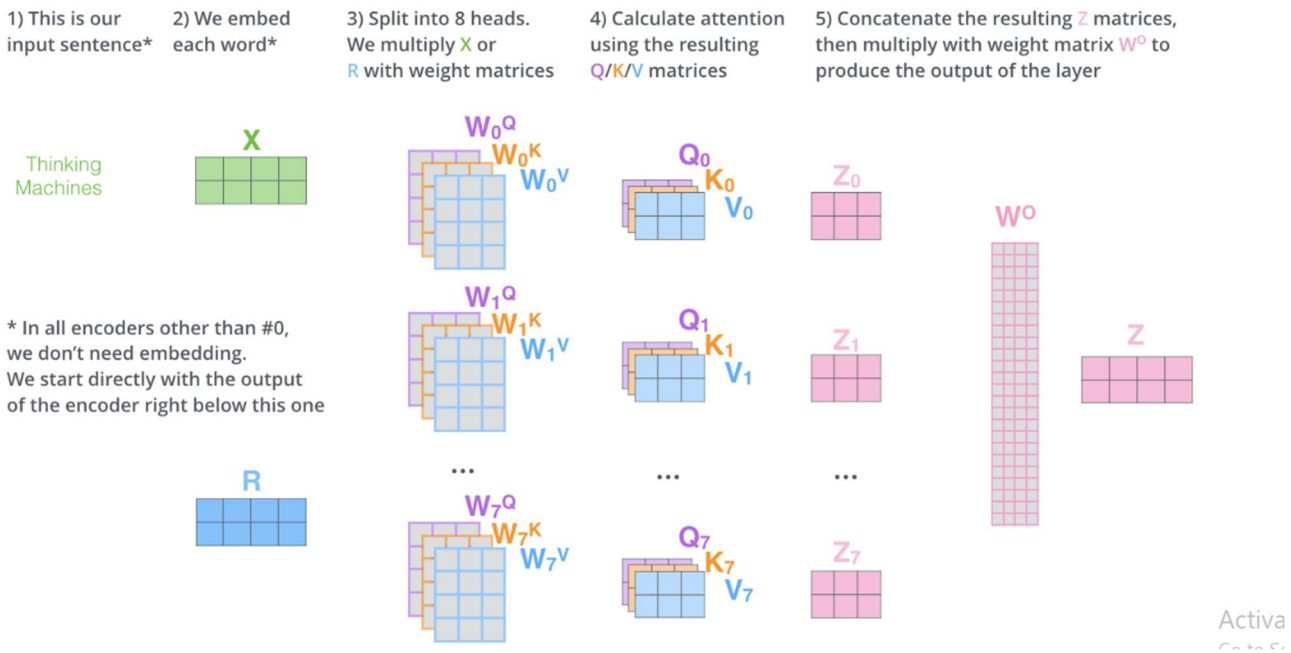
Multi-head Attention

The Multi-Head Attention mechanism computes the attention between each pair of positions in a sequence. It consists of multiple "attention heads" that capture different aspects of the input sequence. Mathematically, it is expressed as:

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

What exactly is happening here?

Q, K, and V are batches of matrices, each with shape (batch_size, seq_length, num_features). Multiplying the query (Q) and key (K) arrays results in a (batch_size, seq_length, seq_length) array, which tells us roughly how important each element in the sequence is. This is the attention of this layer — it determines which elements we "pay attention" to. The attention array is normalized using softmax, so that all of the weights sum to one. (Because we can't pay more than 100% attention, right?) Finally, the attention is applied to the value (V) array using matrix multiplication.



Feed Forward Neural Network

Each of the layers in our encoder and decoder contains a fully connected feed-forward network, which consists of two linear transformations with a ReLU activation in between. The dimensionality of input and output is 512, and the inner-layer has dimensionality 2048.

Positional Encoding

Positional Encoding is used to inject the position information of each token in the input sequence. It uses sine and cosine functions of different frequencies to generate the positional encoding. Vaswani et. al. encode positional information using trigonometric functions, according to the equation:

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

Add & Normalization

In each encoder has a residual connection around it and is followed by a layer-normalization step.

The output of each sub-layer is LayerNorm(x + Sublayer(x)), where Sublayer(x) is the function implemented by the sub-layer itself. We apply dropout to the output of each sub-layer, before it is added to the sub-layer input and normalized.

Lab Exercise to be performed in this session:

1. **Data Preprocessing: Tokenize**, create vocabulary and mapping between words and integer IDs., Convert sentences to numerical sequences., Pad or truncate sequences to a fixed length.
2. **Embeddings:** Initialize word embeddings and positional encodings., Combine word embeddings and positional encodings for each input sequence.
3. **Encoder-Decoder Architecture: The** encoder consists of self-attention layers and feed-forward layers., The decoder includes self-attention, encoder-decoder attention, and feed-forward layers.
4. **Self-Attention Mechanism: Calculate** attention scores for each word in the sequence, Compute weighted sums for context information for each word.
5. **Multi-Head Attention: Extend** self-attention to multi-head attention, Create multiple sets of weights and biases for different attention heads. Concatenate and linearly transform the results from each head.
6. **Positional Encoding: Generate** positional encoding vectors for each position in the sequence. Add positional encodings to word embeddings.
7. **Position-wise Feed-Forward Networks:** Implement position-wise feed-forward networks within the Transformer. Apply a linear transformation followed by a non-linear activation function. The output dimension is typically larger than the input dimension.
8. **Layer Normalization and Residual Connections:** Apply layer normalization after each sub-layer (e.g., self-attention or feed-forward). Use residual connections to add the input to the output of each sub-layer.
9. **Encoder Stack:** Stack multiple encoder layers on top of each other. Each layer takes the output of the previous layer as input.
10. **Decoder Stack:** Stack multiple decoder layers on top of each other. Implement the masking mechanism for ensuring that each position can only attend to positions before it.

NAME: UMANG KIRIT LODAYA

SAP ID: 600009200032

BATCH: D11

In [1]:
```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.r
ead_csv)

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

In [2]:
```
import math
import re
import warnings
import seaborn as sns
import matplotlib.pyplot as plt
```

```
/opt/conda/lib/python3.10/site-packages/scipy/__init__.py:14(
UserWarning: A NumPy version ≥1.16.5 and <1.23.0 is required
for this version of SciPy (detected version 1.23.5
  warnings.warn(f"A NumPy version ≥{np_minversion} and <{np_m
axversion}"
```

In [3]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

In [4]:
```
class Embedding(nn.Module):
    def __init__(self, vocab_size, embedding_dim):
        super(Embedding, self).__init__()
        self.embed = nn.Embedding(vocab_size, embedding_dim)

    def forward(self, text):
        out = self.embed(text)
        return out
```

In [5]:

```
class PositionalEmbedding(nn.Module):
    def __init__(self, max_seq_len, embed_model_dim):
        super(PositionalEmbedding, self).__init__()
        self.embed_dim = embed_model_dim
        pe = torch.zeros(max_seq_len, self.embed_dim)

        for pos in range(max_seq_len):
            for i in range(0, self.embed_dim, 2):
                pe[pos, i] = math.sin(pos / (10000 ** ((2 *
i)/self.embed_dim)))
                pe[pos, i + 1] = math.cos(pos / (10000 ** ((2
* (i + 1))/self.embed_dim)))

        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x * math.sqrt(self.embed_dim)
        seq_len = x.size(1)
        x = x + torch.autograd.Variable(self.pe[:, :seq_len],
requires_grad = False)
        return x
```

In [6]:

```
class MultiHeadAttention(nn.Module):
    def __init__(self, embed_dim = 512, n_heads = 8):
        super(MultiHeadAttention, self).__init__()

        self.embed_dim = embed_dim
        self.n_heads = n_heads
        self.single_head_dim = int(self.embed_dim / self.n_hea
ds)
        #512/8 = 64  . each key,query, value will be of 64d

        #key,query and value matrixes
        #64 x 64
        self.query_matrix = nn.Linear(self.single_head_dim, se
lf.single_head_dim, bias = False)
        self.key_matrix = nn.Linear(self.single_head_dim, sel
f.single_head_dim, bias = False)
        self.value_matrix = nn.Linear(self.single_head_dim, se
lf.single_head_dim, bias = False)
        self.out = nn.Linear(self.embed_dim, self.embed_dim)

    def forward(self, key, query, value, mask = None):
        batch_size = key.size(0)
        seq_length = key.size(1)
        seq_length_query = query.size(1)

        key = key.view(batch_size, seq_length, self.n_heads, s
elf.single_head_dim)
        query = query.view(batch_size, seq_length_query, self.
n_heads, self.single_head_dim)
        value = value.view(batch_size, seq_length, self.n_head
s, self.single_head_dim)

        k = self.key_matrix(key)
        q = self.query_matrix(query)
        v = self.value_matrix(value)

        q = q.transpose(1, 2)
        k = k.transpose(1, 2)
        v = v.transpose(1, 2)

        k_adjusted = k.transpose(-1, -2)
        product = torch.matmul(q, k_adjusted)

        if mask:
             product = product.masked_fill(mask == 0, float("-
1e20"))

        product = product / math.sqrt(self.single_head_dim) #
/ sqrt(64)

        scores = F.softmax(product, dim = -1)

        scores = torch.matmul(scores, v)
        concat = scores.transpose(1, 2).contiguous().view(batc
h_size, seq_length_query, self.single_head_dim*self.n_heads)
        output = self.out(concat) #(32,10,512) -> (32,10,512)
        return output
```

In [7]:
```
class TransformerBlock(nn.Module):
    def __init__(self, embed_dim, expansion_factor=4, n_heads=
8):
        super(TransformerBlock, self).__init__()

        self.attention = MultiHeadAttention(embed_dim, n_head
s)
        self.norm1 = nn.LayerNorm(embed_dim)
        self.norm2 = nn.LayerNorm(embed_dim)
        self.feed_forward = nn.Sequential(
                            nn.Linear(embed_dim, expansion_facto
r*embed_dim),
                            nn.ReLU(),
                            nn.Linear(expansion_factor*embed_di
m, embed_dim)
        )
        self.dropout1 = nn.Dropout(0.2)
        self.dropout2 = nn.Dropout(0.2)

    def forward(self,key,query,value):
        attention_out = self.attention(key,query,value)
        attention_residual_out = attention_out + value
        norm1_out = self.dropout1(self.norm1(attention_residua
l_out))

        feed_fwd_out = self.feed_forward(norm1_out)
        feed_fwd_residual_out = feed_fwd_out + norm1_out
        norm2_out = self.dropout2(self.norm2(feed_fwd_residual
_out))

        return norm2_out
```

In [8]:
```
class TransformerEncoder(nn.Module):
    def __init__(self, seq_len, vocab_size, embed_dim, num_lay
ers = 2, expansion_factor = 4, n_heads = 8):
        super(TransformerEncoder, self).__init__()

        self.embedding_layer = Embedding(vocab_size, embed_di
m)
        self.positional_encoder = PositionalEmbedding(seq_len,
embed_dim)

        self.layers = nn.ModuleList([TransformerBlock(embed_di
m, expansion_factor, n_heads) for i in range(num_layers)])

    def forward(self, x):
        embed_out = self.embedding_layer(x)
        out = self.positional_encoder(embed_out)
        for layer in self.layers:
            out = layer(out,out,out)

        return out
```

In [9]:
```
class DecoderBlock(nn.Module):
    def __init__(self, embed_dim, expansion_factor=4, n_heads=
8):
        super(DecoderBlock, self).__init__()
        self.attention = MultiHeadAttention(embed_dim, n_heads
=8)
        self.norm = nn.LayerNorm(embed_dim)
        self.dropout = nn.Dropout(0.2)
        self.transformer_block = TransformerBlock(embed_dim, e
xpansion_factor, n_heads)

    def forward(self, key, query, x,mask):
        #we need to pass mask mask only to fst attention
        attention = self.attention(x,x,x,mask=mask) #32x10x512
        value = self.dropout(self.norm(attention + x))
        out = self.transformer_block(key, query, value)
        return out
```

In [10]:
```
class TransformerDecoder(nn.Module):
    def __init__(self, target_vocab_size, embed_dim, seq_len,
num_layers=2, expansion_factor=4, n_heads=8):
        super(TransformerDecoder, self).__init__()
        self.word_embedding = nn.Embedding(target_vocab_size,
embed_dim)
        self.position_embedding = PositionalEmbedding(seq_len,
embed_dim)

        self.layers = nn.ModuleList(
            [
                DecoderBlock(embed_dim, expansion_factor=4, n_
heads=8)
                for _ in range(num_layers)
            ]

        )
        self.fc_out = nn.Linear(embed_dim, target_vocab_size)
        self.dropout = nn.Dropout(0.2)

    def forward(self, x, enc_out, mask):
        x = self.word_embedding(x)  #32x10x512
        x = self.position_embedding(x) #32x10x512
        x = self.dropout(x)
        for layer in self.layers:
            x = layer(enc_out, x, enc_out, mask)

        out = F.softmax(self.fc_out(x))
        return out
```

In [11]:
```
class Transformer(nn.Module):
    def __init__(self, embed_dim, src_vocab_size, target_vocab
_size, seq_length,num_layers=2, expansion_factor=4, n_heads=
8):
        super(Transformer, self).__init__()
        self.target_vocab_size = target_vocab_size
        self.encoder = TransformerEncoder(seq_length, src_voca
b_size, embed_dim, num_layers=num_layers, expansion_factor=exp
ansion_factor, n_heads=n_heads)
        self.decoder = TransformerDecoder(target_vocab_size, e
mbed_dim, seq_length, num_layers=num_layers, expansion_factor=
expansion_factor, n_heads=n_heads)

    def make_trg_mask(self, trg):
        batch_size, trg_len = trg.shape
        # returns the lower triangular part of matrix filled w
ith ones
        trg_mask = torch.tril(torch.ones((trg_len, trg_len))).
expand(
            batch_size, 1, trg_len, trg_len
        )
        return trg_mask

    def decode(self,src,trg):
        trg_mask = self.make_trg_mask(trg)
        enc_out = self.encoder(src)
        out_labels = []
        batch_size,seq_len = src.shape[0],src.shape[1]
        #outputs = torch.zeros(seq_len, batch_size, self.targe
t_vocab_size)
        out = trg
        for i in range(seq_len): #10
            out = self.decoder(out,enc_out,trg_mask) #bs x seq
_len x vocab_dim
            # taking the last token
            out = out[:,-1,:]
            out = out.argmax(-1)
            out_labels.append(out.item())
            out = torch.unsqueeze(out,axis=0)

        return out_labels

    def forward(self, src, trg):
        trg_mask = self.make_trg_mask(trg)
        enc_out = self.encoder(src)
        outputs = self.decoder(trg, enc_out, trg_mask)
        return outputs
```

In [12]:
```
src_vocab_size = 11
target_vocab_size = 11
num_layers = 6
seq_length= 12
```

In [13]:
```
model = Transformer(embed_dim=512, src_vocab_size=src_vocab_
ze,
                    target_vocab_size=target_vocab_size, seq_l
ength=seq_length,
                    num_layers=num_layers, expansion_factor=4,
n_heads=8)

model
```

Out[13]:
```
Transformer(
  (encoder): TransformerEncoder(
    (embedding_layer): Embedding(
      (embed): Embedding(11, 512)
    )
    (positional_encoder): PositionalEmbedding()
    (layers): ModuleList(
      (0-5): 6 x TransformerBlock(
        (attention): MultiHeadAttention(
          (query_matrix): Linear(in_features=64, out_features=
64, bias=False)
          (key_matrix): Linear(in_features=64, out_features=6
4, bias=False)
          (value_matrix): Linear(in_features=64, out_features=
64, bias=False)
          (out): Linear(in_features=512, out_features=512, bia
s=True)
        )
        (norm1): LayerNorm((512,), eps=1e-05, elementwise_affi
ne=True)
        (norm2): LayerNorm((512,), eps=1e-05, elementwise_affi
ne=True)
        (feed_forward): Sequential(
          (0): Linear(in_features=512, out_features=2048, bias
=True)
          (1): ReLU()
          (2): Linear(in_features=2048, out_features=512, bias
=True)
        )
        (dropout1): Dropout(p=0.2, inplace=False)
        (dropout2): Dropout(p=0.2, inplace=False)
      )
    )
  )
  (decoder): TransformerDecoder(
    (word_embedding): Embedding(11, 512)
    (position_embedding): PositionalEmbedding()
    (layers): ModuleList(
      (0-5): 6 x DecoderBlock(
        (attention): MultiHeadAttention(
          (query_matrix): Linear(in_features=64, out_features=
64, bias=False)
          (key_matrix): Linear(in_features=64, out_features=6
4, bias=False)
          (value_matrix): Linear(in_features=64, out_features=
64, bias=False)
          (out): Linear(in_features=512, out_features=512, bia
s=True)
        )
        (norm): LayerNorm((512,), eps=1e-05, elementwise_affin
e=True)
        (dropout): Dropout(p=0.2, inplace=False)
        (transformer_block): TransformerBlock(
          (attention): MultiHeadAttention(
            (query_matrix): Linear(in_features=64, out_feature
s=64, bias=False)
            (key_matrix): Linear(in_features=64, out_features=
64, bias=False)
            (value_matrix): Linear(in_features=64, out_feature
s=64, bias=False)
            (out): Linear(in_features=512, out_features=512, b
```

```
ias=True)
          )
          (norm1): LayerNorm((512,), eps=1e-05, elementwise_af
fine=True)
          (norm2): LayerNorm((512,), eps=1e-05, elementwise_af
fine=True)
          (feed_forward): Sequential(
            (0): Linear(in_features=512, out_features=2048, bi
as=True)
            (1): ReLU()
            (2): Linear(in_features=2048, out_features=512, bi
as=True)
          )
          (dropout1): Dropout(p=0.2, inplace=False)
          (dropout2): Dropout(p=0.2, inplace=False)
        )
      )
    )
    (fc_out): Linear(in_features=512, out_features=11, bias=Tr
ue)
    (dropout): Dropout(p=0.2, inplace=False)
  )
)
```

In [14]:
```
src = torch.tensor([[0, 2, 5, 6, 4, 3, 9, 5, 2, 9, 10, 1]])
trg = torch.tensor([[0]])
print(src.shape, trg.shape)
out = model.decode(src, trg)
out
```

```
torch.Size([1, 12]) torch.Size([1, 1])

/tmp/ipykernel_32/1089410970.py:24: UserWarning: Implicit di
nsion choice for softmax has been deprecated. Change the call
to include dim=X as an argument.
  out = F.softmax(self.fc_out(x))
```

Out[14]:
```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```