Sahil Y. Tike
BE B3 (212066)

Transformers From Scratch Using Pytorch

Type *Markdown* and LaTeX: α^2

1. Introduction

Implement transformers in "Attention is all you need paper" from scratch using Pytorch. Basically transformer have an encoder-decoder architecture. It is common for language translation models.

2. Import Libraries

In [ ]:
```
pip install torchvision
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: torchvision in /home/dipali/.local/lib/python
3.8/site-packages (0.15.1)
Collecting torch==2.0.0
Note: you may need to restart the kernel to use updated packages.
```

In [ ]:
```
pip install torchtext==0.10.0
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: torchtext==0.10.0 in /home/dipali/.local/lib/p
ython3.8/site-packages (0.10.0)
Requirement already satisfied: torch==1.9.0 in /home/dipali/.local/lib/python
3.8/site-packages (from torchtext==0.10.0) (1.9.0)
Requirement already satisfied: numpy in /home/dipali/.local/lib/python3.8/sit
e-packages (from torchtext==0.10.0) (1.24.1)
Requirement already satisfied: requests in /usr/lib/python3/dist-packages (fr
om torchtext==0.10.0) (2.22.0)
Requirement already satisfied: tqdm in /home/dipali/.local/lib/python3.8/site
-packages (from torchtext==0.10.0) (4.64.1)
Requirement already satisfied: typing-extensions in /home/dipali/.local/lib/p
ython3.8/site-packages (from torch==1.9.0->torchtext==0.10.0) (4.4.0)

[notice] A new release of pip is available: 23.0 -> 23.0.1
[notice] To update, run: python3 -m pip install --upgrade pip
Note: you may need to restart the kernel to use updated packages.
```

]

In [ ]:
```python
# importing required libraries
import torch.nn as nn
import torch
import torch.nn.functional as F
import math,copy,re
import warnings
import pandas as pd
import numpy as np
import seaborn as sns
import torchtext
import matplotlib.pyplot as plt
warnings.simplefilter("ignore")
print(torch.__version__)
```

```
1.9.0+cu102
```

3. Basic components

In [ ]:
```python
class Embedding(nn.Module):
    def __init__(self, vocab_size, embed_dim):
        """
        Args:
            vocab_size: size of vocabulary
            embed_dim: dimension of embeddings
        """
        super(Embedding, self).__init__()
        self.embed = nn.Embedding(vocab_size, embed_dim)
    def forward(self, x):
        """
        Args:
            x: input vector
        Returns:
            out: embedding vector
        """
        out = self.embed(x)
        return out
```

]

In [ ]:
```
# register buffer in Pytorch ->
# If you have parameters in your model, which should be saved and restored in
# but not trained by the optimizer, you should register them as buffers.


class PositionalEmbedding(nn.Module):
    def __init__(self,max_seq_len,embed_model_dim):
        """
        Args:
            seq_len: length of input sequence
            embed_model_dim: demension of embedding
        """
        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.
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)


    def forward(self, x):
        """
        Args:
            x: input vector
        Returns:
            x: output
        """

        # make embeddings relatively larger
        x = x * math.sqrt(self.embed_dim)
        #add constant to embedding
        seq_len = x.size(1)
        x = x + torch.autograd.Variable(self.pe[:,:seq_len], requires_grad=Fal
        return x
```

In [ ]:
```
class MultiHeadAttention(nn.Module):
    def __init__(self, embed_dim=512, n_heads=8):
        """
        Args:
            embed_dim: dimension of embeding vector output
            n_heads: number of self attention heads
        """
        super(MultiHeadAttention, self).__init__()

        self.embed_dim = embed_dim    #512 dim
        self.n_heads = n_heads   #8
        self.single_head_dim = int(self.embed_dim / self.n_heads)   #512/8 = 6

        #key,query and value matrixes    #64 x 64
        self.query_matrix = nn.Linear(self.single_head_dim , self.single_head_
        self.key_matrix = nn.Linear(self.single_head_dim  , self.single_head_c
        self.value_matrix = nn.Linear(self.single_head_dim ,self.single_head_c
        self.out = nn.Linear(self.n_heads*self.single_head_dim ,self.embed_dim

    def forward(self,key,query,value,mask=None):    #batch_size x sequence_len

        """
        Args:
           key : key vector
           query : query vector
           value : value vector
           mask: mask for decoder

        Returns:
           output vector from multihead attention
        """
        batch_size = key.size(0)
        seq_length = key.size(1)

        # query dimension can change in decoder during inference.
        # so we cant take general seq_length
        seq_length_query = query.size(1)

        # 32x10x512
        key = key.view(batch_size, seq_length, self.n_heads, self.single_head_
        query = query.view(batch_size, seq_length_query, self.n_heads, self.si
        value = value.view(batch_size, seq_length, self.n_heads, self.single_h

        k = self.key_matrix(key)       # (32x10x8x64)
        q = self.query_matrix(query)
        v = self.value_matrix(value)

        q = q.transpose(1,2)  # (batch_size, n_heads, seq_len, single_head_dim
        k = k.transpose(1,2)  # (batch_size, n_heads, seq_len, single_head_dim
        v = v.transpose(1,2)  # (batch_size, n_heads, seq_len, single_head_dim

        # computes attention
        # adjust key for matrix multiplication
        k_adjusted = k.transpose(-1,-2)  #(batch_size, n_heads, single_head_di
        product = torch.matmul(q, k_adjusted)  #(32 x 8 x 10 x 64) x (32 x 8 x
```

```
        # fill those positions of product matrix as (-1e20) where mask positio
        if mask is not None:
             product = product.masked_fill(mask == 0, float("-1e20"))

        #divising by square root of key dimension
        product = product / math.sqrt(self.single_head_dim) # / sqrt(64)

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

        #mutiply with value matrix
        scores = torch.matmul(scores, v)  ##(32x8x 10x 10) x (32 x 8 x 10 x 64

        #concatenated output
        concat = scores.transpose(1,2).contiguous().view(batch_size, seq_lengt

        output = self.out(concat) #(32,10,512) -> (32,10,512)

        return output
```

4. Encoder

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

                """
                Args:
                   embed_dim: dimension of the embedding
                   expansion_factor: fator ehich determines output dimension of linear
                   n_heads: number of attention heads

                """
                self.attention = MultiHeadAttention(embed_dim, n_heads)

                self.norm1 = nn.LayerNorm(embed_dim)
                self.norm2 = nn.LayerNorm(embed_dim)

                self.feed_forward = nn.Sequential(
                                  nn.Linear(embed_dim, expansion_factor*embed_dim),
                                  nn.ReLU(),
                                  nn.Linear(expansion_factor*embed_dim, embed_dim)
                )

                self.dropout1 = nn.Dropout(0.2)
                self.dropout2 = nn.Dropout(0.2)

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

                """
                Args:
                   key: key vector
                   query: query vector
                   value: value vector
                   norm2_out: output of transformer block

                """

                attention_out = self.attention(key,query,value)  #32x10x512
                attention_residual_out = attention_out + value  #32x10x512
                norm1_out = self.dropout1(self.norm1(attention_residual_out)) #32x10x5

                feed_fwd_out = self.feed_forward(norm1_out) #32x10x512 -> #32x10x2048
                feed_fwd_residual_out = feed_fwd_out + norm1_out #32x10x512
                norm2_out = self.dropout2(self.norm2(feed_fwd_residual_out)) #32x10x51

                return norm2_out



        class TransformerEncoder(nn.Module):
            """
            Args:
                seq_len : length of input sequence
                embed_dim: dimension of embedding
                num_layers: number of encoder layers
                expansion_factor: factor which determines number of linear layers in f
                n_heads: number of heads in multihead attention
```

```
    Returns:
        out: output of the encoder
    """
    def __init__(self, seq_len, vocab_size, embed_dim, num_layers=2, expansior
        super(TransformerEncoder, self).__init__()

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

        self.layers = nn.ModuleList([TransformerBlock(embed_dim, expansion_fac

    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  #32x10x512
```

5. Decoder

In [ ]:

```python
class DecoderBlock(nn.Module):
    def __init__(self, embed_dim, expansion_factor=4, n_heads=8):
        super(DecoderBlock, self).__init__()

        """
        Args:
           embed_dim: dimension of the embedding
           expansion_factor: fator ehich determines output dimension of linear
           n_heads: number of attention heads

        """
        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, expansion_factor,


    def forward(self, key, query, x,mask):

        """
        Args:
           key: key vector
           query: query vector
           value: value vector
           mask: mask to be given for multi head attention
        Returns:
           out: output of transformer block

        """

        #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


class TransformerDecoder(nn.Module):
    def __init__(self, target_vocab_size, embed_dim, seq_len, num_layers=2, ex
        super(TransformerDecoder, self).__init__()
        """
        Args:
           target_vocab_size: vocabulary size of taget
           embed_dim: dimension of embedding
           seq_len : length of input sequence
           num_layers: number of encoder layers
           expansion_factor: factor which determines number of linear layers i
           n_heads: number of heads in multihead attention

        """
        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):

        """
        Args:
            x: input vector from target
            enc_out : output from encoder layer
            trg_mask: mask for decoder self attention
        Returns:
            out: output vector
        """


        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
```

Finally we will arrange all submodules and creates the entire tranformer architecture.

In [ ]:

```
class Transformer(nn.Module):
    def __init__(self, embed_dim, src_vocab_size, target_vocab_size, seq_lengt
        super(Transformer, self).__init__()

        """
        Args:
           embed_dim:  dimension of embedding
           src_vocab_size: vocabulary size of source
           target_vocab_size: vocabulary size of target
           seq_length : length of input sequence
           num_layers: number of encoder layers
           expansion_factor: factor which determines number of linear layers i
           n_heads: number of heads in multihead attention

        """

        self.target_vocab_size = target_vocab_size

        self.encoder = TransformerEncoder(seq_length, src_vocab_size, embed_di
        self.decoder = TransformerDecoder(target_vocab_size, embed_dim, seq_le


    def make_trg_mask(self, trg):
        """
        Args:
            trg: target sequence
        Returns:
            trg_mask: target mask
        """
        batch_size, trg_len = trg.shape
        # returns the lower triangular part of matrix filled with 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):
        """
        for inference
        Args:
            src: input to encoder
            trg: input to decoder
        out:
            out_labels : returns final prediction of sequence
        """
        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.target_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,:]
```

```python
            out = out.argmax(-1)
            out_labels.append(out.item())
            out = torch.unsqueeze(out,axis=0)


        return out_labels

    def forward(self, src, trg):
        """
        Args:
            src: input to encoder
            trg: input to decoder
        out:
            out: final vector which returns probabilities of each target word
        """
        trg_mask = self.make_trg_mask(trg)
        enc_out = self.encoder(src)

        outputs = self.decoder(trg, enc_out, trg_mask)
        return outputs
```

6. Testing Code

Suppose we have input sequence oflength 10 and target sequence of length 10.

In [ ]:

```
src_vocab_size = 11
target_vocab_size = 11
num_layers = 6
seq_length= 12


# let 0 be sos token and 1 be eos token
src = torch.tensor([[0, 2, 5, 6, 4, 3, 9, 5, 2, 9, 10, 1],
                    [0, 2, 8, 7, 3, 4, 5, 6, 7, 2, 10, 1]])
target = torch.tensor([[0, 1, 7, 4, 3, 5, 9, 2, 8, 10, 9, 1],
                       [0, 1, 5, 6, 2, 4, 7, 6, 2, 8, 10, 1]])

print(src.shape,target.shape)
model = Transformer(embed_dim=512, src_vocab_size=src_vocab_size,
                    target_vocab_size=target_vocab_size, seq_length=seq_length
                    num_layers=num_layers, expansion_factor=4, n_heads=8)
model
```

```
torch.Size([2, 12]) torch.Size([2, 12])
```

Out[10]:

```
Transformer(
  (encoder): TransformerEncoder(
    (embedding_layer): Embedding(
      (embed): Embedding(11, 512)
    )
    (positional_encoder): PositionalEmbedding()
    (layers): ModuleList(
      (0): TransformerBlock(
        (attention): MultiHeadAttention(
          (query_matrix): Linear(in_features=64, out_features=64, bias=Fal
se)
          (key_matrix): Linear(in_features=64, out_features=64, bias=Fals
e)
          (value_matrix): Linear(in_features=64, out_features=64, bias=Fal
se)
          (out): Linear(in_features=512, out_features=512, bias=True)
        )
        (norm1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
```

In [ ]:

```
out = model(src, target)
out.shape
```

Out[11]:

```
torch.Size([2, 12, 11])
```

In [ ]:

```python
# inference
model = Transformer(embed_dim=512, src_vocab_size=src_vocab_size,
                    target_vocab_size=target_vocab_size, seq_length=seq_length
                    num_layers=num_layers, expansion_factor=4, n_heads=8)



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])
```

Out[12]:

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```