Name: Onasvee Banarse

Class: BE Computer-1

Roll No: 09

Problem Statement: Create a transformer from scratch using the Pytorch library

Libs

In [1]:
```
from torch.utils.data import Dataset
import torch.nn.functional as F
from collections import Counter
from os.path import exists
import torch.optim as optim
import torch.nn as nn
import numpy as np
import random
import torch
import math
import re
```

Transformer

In [2]:
```
def attention(q, k, v, mask = None, dropout = None):
    scores = q.matmul(k.transpose(-2, -1))
    scores /= math.sqrt(q.shape[-1])

    #mask
    scores = scores if mask is None else scores.masked_fill(mask == 0, -1e3)

    scores = F.softmax(scores, dim = -1)
    scores = dropout(scores) if dropout is not None else scores
    output = scores.matmul(v)
    return output

class MultiHeadAttention(nn.Module):
    def __init__(self, n_heads, out_dim, dropout=0.1):
        super().__init__()

#        self.q_linear = nn.Linear(out_dim, out_dim)
#        self.k_linear = nn.Linear(out_dim, out_dim)
#        self.v_linear = nn.Linear(out_dim, out_dim)
        self.linear = nn.Linear(out_dim, out_dim*3)

        self.n_heads = n_heads
        self.out_dim = out_dim
        self.out_dim_per_head = out_dim // n_heads
        self.out = nn.Linear(out_dim, out_dim)
        self.dropout = nn.Dropout(dropout)

    def split_heads(self, t):
```

```
        return t.reshape(t.shape[0], -1, self.n_heads, self.out_dim_per_head)

    def forward(self, x, y=None, mask=None):
        #in decoder, y comes from encoder. In encoder, y=x
        y = x if y is None else y

        qkv = self.linear(x) # BS * SEQ_LEN * (3*EMBED_SIZE_L)
        q = qkv[:, :, :self.out_dim] # BS * SEQ_LEN * EMBED_SIZE_L
        k = qkv[:, :, self.out_dim:self.out_dim*2] # BS * SEQ_LEN * EMBED_SIZE_L
        v = qkv[:, :, self.out_dim*2:] # BS * SEQ_LEN * EMBED_SIZE_L

        #break into n_heads
        q, k, v = [self.split_heads(t) for t in (q,k,v)]  # BS * SEQ_LEN * HEAD * E
        q, k, v = [t.transpose(1,2) for t in (q,k,v)]  # BS * HEAD * SEQ_LEN * EMBE

        #n_heads => attention => merge the heads => mix information
        scores = attention(q, k, v, mask, self.dropout) # BS * HEAD * SEQ_LEN * EMB
        scores = scores.transpose(1,2).contiguous().view(scores.shape[0], -1, self.
        out = self.out(scores)  # BS * SEQ_LEN * EMBED_SIZE

        return out

class FeedForward(nn.Module):
    def __init__(self, inp_dim, inner_dim, dropout=0.1):
        super().__init__()
        self.linear1 = nn.Linear(inp_dim, inner_dim)
        self.linear2 = nn.Linear(inner_dim, inp_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        #inp => inner => relu => dropout => inner => inp
        return self.linear2(self.dropout(F.relu(self.linear1(x))))

class EncoderLayer(nn.Module):
    def __init__(self, n_heads, inner_transformer_size, inner_ff_size, dropout=0.1)
        super().__init__()
        self.mha = MultiHeadAttention(n_heads, inner_transformer_size, dropout)
        self.ff = FeedForward(inner_transformer_size, inner_ff_size, dropout)
        self.norm1 = nn.LayerNorm(inner_transformer_size)
        self.norm2 = nn.LayerNorm(inner_transformer_size)
        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        x2 = self.norm1(x)
        x = x + self.dropout1(self.mha(x2, mask=mask))
        x2 = self.norm2(x)
        x = x + self.dropout2(self.ff(x2))
        return x

class Transformer(nn.Module):
    def __init__(self, n_code, n_heads, embed_size, inner_ff_size, n_embeddings, se
        super().__init__()

        #model input
        self.embeddings = nn.Embedding(n_embeddings, embed_size)
        self.pe = PositionalEmbedding(embed_size, seq_len)

        #backbone
        encoders = []
        for i in range(n_code):
```

```
            encoders += [EncoderLayer(n_heads, embed_size, inner_ff_size, dropout)]
        self.encoders = nn.ModuleList(encoders)

        #language model
        self.norm = nn.LayerNorm(embed_size)
        self.linear = nn.Linear(embed_size, n_embeddings, bias=False)


    def forward(self, x):
        x = self.embeddings(x)
        x = x + self.pe(x)
        for encoder in self.encoders:
            x = encoder(x)
        x = self.norm(x)
        x = self.linear(x)
        return x
```

Positional Embedding

In [3]:
```
class PositionalEmbedding(nn.Module):
    def __init__(self, d_model, max_seq_len = 80):
        super().__init__()
        self.d_model = d_model
        pe = torch.zeros(max_seq_len, d_model)
        pe.requires_grad = False
        for pos in range(max_seq_len):
            for i in range(0, d_model, 2):
                pe[pos, i] = math.sin(pos / (10000 ** ((2 * i)/d_model)))
                pe[pos, i + 1] = math.cos(pos / (10000 ** ((2 * (i + 1))/d_model)))
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        return self.pe[:,:x.size(1)] #x.size(1) = seq_len
```

Dataset

In [4]:
```
class SentencesDataset(Dataset):
    #Init dataset
    def __init__(self, sentences, vocab, seq_len):
        dataset = self

        dataset.sentences = sentences
        dataset.vocab = vocab + ['<ignore>', '<oov>', '<mask>']
        dataset.vocab = {e:i for i, e in enumerate(dataset.vocab)}
        dataset.rvocab = {v:k for k,v in dataset.vocab.items()}
        dataset.seq_len = seq_len

        #special tags
        dataset.IGNORE_IDX = dataset.vocab['<ignore>'] #replacement tag for tokens
        dataset.OUT_OF_VOCAB_IDX = dataset.vocab['<oov>'] #replacement tag for unkn
        dataset.MASK_IDX = dataset.vocab['<mask>'] #replacement tag for the masked


    #fetch data
    def __getitem__(self, index, p_random_mask=0.15):
        dataset = self
```

```
        #while we don't have enough word to fill the sentence for a batch
        s = []
        while len(s) < dataset.seq_len:
            s.extend(dataset.get_sentence_idx(index % len(dataset)))
            index += 1

        #ensure that the sequence is of length seq_len
        s = s[:dataset.seq_len]
        [s.append(dataset.IGNORE_IDX) for i in range(dataset.seq_len - len(s))] #PA

        #apply random mask
        s = [(dataset.MASK_IDX, w) if random.random() < p_random_mask else (w, data

        return {'input': torch.Tensor([w[0] for w in s]).long(),
                'target': torch.Tensor([w[1] for w in s]).long()}

    #return length
    def __len__(self):
        return len(self.sentences)

    #get words id
    def get_sentence_idx(self, index):
        dataset = self
        s = dataset.sentences[index]
        s = [dataset.vocab[w] if w in dataset.vocab else dataset.OUT_OF_VOCAB_IDX f
        return s
```

Methods / Class

In [5]:

```
def get_batch(loader, loader_iter):
    try:
        batch = next(loader_iter)
    except StopIteration:
        loader_iter = iter(loader)
        batch = next(loader_iter)
    return batch, loader_iter
```

Initialization

In [6]:

```
print('initializing..')
batch_size = 128
seq_len = 20
embed_size = 128
inner_ff_size = embed_size * 4
n_heads = 8
n_code = 8
n_vocab = 40000
dropout = 0.1
n_workers = 12

#optimizer
optim_kwargs = {'lr':2e-3, 'weight_decay':1e-4, 'betas':(.9,.999)}
```

```
initializing..
```

Input

```
In [7]: #1) Load text
print('loading text...')
pth = 'europarl30k.fr.txt'
sentences = open(pth, encoding='utf-8').read().lower().split('\n')

#2) tokenize sentences (can be done during training, you can also use spacy udpipe)
print('tokenizing sentences...')
special_chars = ',?;.:/*!+-()[]{}"\'&'
sentences = [re.sub(f'[{re.escape(special_chars)}]', ' \g<0> ', s).split(' ') for s
sentences = [[w for w in s if len(w)] for s in sentences]

#3) create vocab if not already created
print('creating/loading vocab...')
pth = 'vocab.txt'
if not exists(pth):
    words = [w for s in sentences for w in s]
    vocab = Counter(words).most_common(n_vocab) #keep the N most frequent words
    vocab = [w[0] for w in vocab]
    open(pth, 'w+').write('\n'.join(vocab))
else:
    vocab = open(pth).read().split('\n')

#4) create dataset
print('creating dataset...')
dataset = SentencesDataset(sentences, vocab, seq_len)
kwargs = {'num_workers':n_workers, 'shuffle':True,  'drop_last':True, 'pin_memory':
data_loader = torch.utils.data.DataLoader(dataset, **kwargs)
```

```
loading text...
tokenizing sentences...
creating/loading vocab...
creating dataset...
```

Model

```
In [8]: print('initializing model...')
model = Transformer(n_code, n_heads, embed_size, inner_ff_size, len(dataset.vocab),
model = model.cuda()
```

```
initializing model...
```

Optimizer

```
In [9]: print('initializing optimizer and loss...')
optimizer = optim.Adam(model.parameters(), **optim_kwargs)
loss_model = nn.CrossEntropyLoss(ignore_index=dataset.IGNORE_IDX)
```

```
initializing optimizer and loss...
```

Train

```
In [10]: print('training...')
print_each = 1000
model.train()
batch_iter = iter(data_loader)
n_iteration = 30000
for it in range(n_iteration):
```

```
    #get batch
    batch, batch_iter = get_batch(data_loader, batch_iter)

    #infer
    masked_input = batch['input']
    masked_target = batch['target']

    masked_input = masked_input.cuda(non_blocking=True)
    masked_target = masked_target.cuda(non_blocking=True)
    output = model(masked_input)

    #compute the cross entropy loss
    output_v = output.view(-1,output.shape[-1])
    target_v = masked_target.view(-1,1).squeeze()
    loss = loss_model(output_v, target_v)

    #compute gradients
    loss.backward()

    #apply gradients
    optimizer.step()

    #print step
    if it % print_each == 0:
        print('it:', it,
              ' | loss', np.round(loss.item(),2),
              ' | Δw:', round(model.embeddings.weight.grad.abs().sum().item(),3))

    #reset gradients
    optimizer.zero_grad()
```

```
training...
it: 0  | loss 10.29  | Δw: 1.389
it: 1000  | loss 4.34  | Δw: 19.957
it: 2000  | loss 3.82  | Δw: 34.516
it: 3000  | loss 3.63  | Δw: 44.884
it: 4000  | loss 3.13  | Δw: 50.024
it: 5000  | loss 3.38  | Δw: 57.732
it: 6000  | loss 3.5  | Δw: 59.555
it: 7000  | loss 3.4  | Δw: 62.795
it: 8000  | loss 3.0  | Δw: 63.495
it: 9000  | loss 3.19  | Δw: 75.824
it: 10000  | loss 3.18  | Δw: 73.606
it: 11000  | loss 2.87  | Δw: 76.833
it: 12000  | loss 3.0  | Δw: 80.099
it: 13000  | loss 2.97  | Δw: 73.402
it: 14000  | loss 3.01  | Δw: 83.089
it: 15000  | loss 3.19  | Δw: 81.15
it: 16000  | loss 2.89  | Δw: 85.772
it: 17000  | loss 2.83  | Δw: 81.786
it: 18000  | loss 2.79  | Δw: 84.248
it: 19000  | loss 2.77  | Δw: 89.305
it: 20000  | loss 2.74  | Δw: 84.135
it: 21000  | loss 3.03  | Δw: 84.263
it: 22000  | loss 2.73  | Δw: 81.011
it: 23000  | loss 2.85  | Δw: 90.095
it: 24000  | loss 2.9  | Δw: 87.738
it: 25000  | loss 3.0  | Δw: 95.965
it: 26000  | loss 2.84  | Δw: 94.525
it: 27000  | loss 2.85  | Δw: 88.609
it: 28000  | loss 2.58  | Δw: 86.334
it: 29000  | loss 2.91  | Δw: 88.332
it: 29995  | loss 2.99  | Δw: 94.651
```

Results analysis

In [11]:

```
print('saving embeddings...')
N = 3000
np.savetxt('values.tsv', np.round(model.embeddings.weight.detach().cpu().numpy()[0
s = [dataset.rvocab[i] for i in range(N)]
open('names.tsv', 'w+').write('\n'.join(s))
```

```
saving embeddings...
```

Out[11]:

```
25394
```