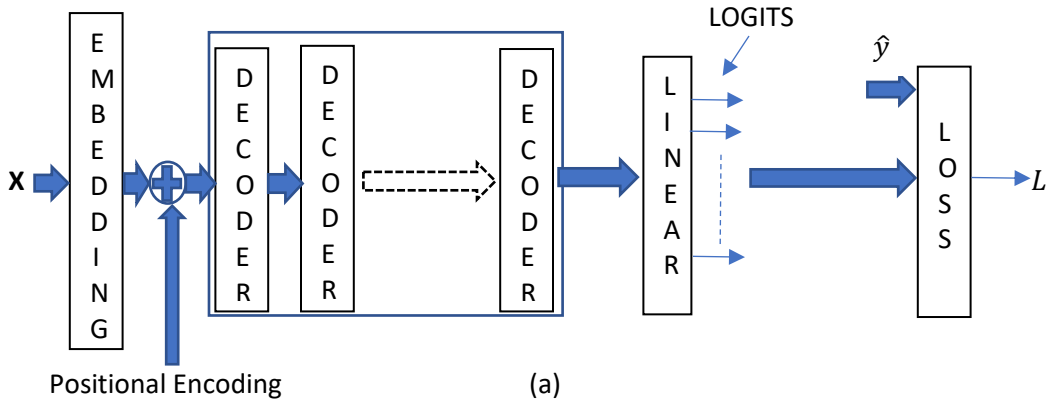
CPEG 592 – Assignment #3 - 2024

Simple Transformer Implementation

Create a Visual Studio project called TransformerXY (replace XY with your first name, last name initials, e.g., I named my project as TransformerAM. Rename the TransformerAM.py to TransformerAMMain.py. Add a folder called data to the project. Then add the enwik8.gz file to this folder. This file is available on the CPEG 589 web site.

We will build a character level Language Model based on the Decoder GPT style architecture as shown below.



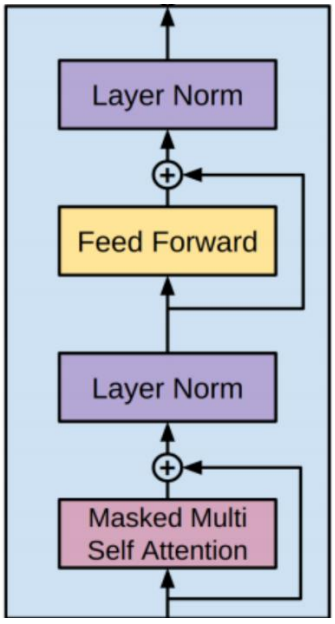


Figure 1. (a) Architecture of GPT Style Transformer. (b) Design of each Transformer Block (Decoder)

The above architecture will be implemented in a clear Object-Oriented style in Pytorch. You will use the Enwik8 dataset to compute the BPC on the test part of the Enwik8 dataset. Recall that Enwik8 has 100 million characters, out of which 90 million will be used for training, 5 million for validation and the rest 5 million for testing.

Add a python class to the project called MyNLPDataSet.py with the following code in it. The dataset loads the data and in the getitem returns seq_len number of characters from a random position in the dataset.

```
import torch
from torch.utils.data import Dataset

class MyNLPDataSet(Dataset):
    def __init__(self, data, seq_len):
        super().__init__()
        self.data = data
        self.seq_len = seq_len

    def __getitem__(self, index):
        rand_start = torch.randint(0, self.data.size(0) - self.seq_len - 1,
(1,))
        full_seq = self.data[rand_start: rand_start + self.seq_len + 1].long()
        return full_seq.cuda()

    def __len__(self):
        return self.data.size(0) // self.seq_len
```

Add a python file to the project called Utils.py with the following code in it. This code has a function called get_loaders_enwiki8 that returns training dataset, the validation dataset, and the training dataloader.

```
import numpy as np
import torch
from MyNLPDataSet import MyNLPDataSet
from torch.utils.data import DataLoader
import gzip

def cycle(loader):
    while True:
        for data in loader:
            yield data

def get_loaders_enwiki8(seq_len, batch_size):
    # ---------prepare enwik8 data-----------
    with gzip.open('./data/enwik8.gz') as file:
        data = np.fromstring(file.read(int(95e6)), dtype = np.uint8)
        data_train, data_val = map(torch.from_numpy, np.split(data,
[int(90e6)]))

    train_dataset = MyNLPDataSet(data_train, seq_len)
```

```
    val_dataset   = MyNLPDataSet(data_val, seq_len)
    train_loader  = cycle(DataLoader(train_dataset, batch_size = batch_size))
    val_loader    = cycle(DataLoader(val_dataset, batch_size = batch_size))
    return train_loader, val_loader, val_dataset
```

Add a python class to the project called PositionalEncoding with the following code in it. It implements the original positional encoding as proposed in the Transformer paper.

```
import torch
from torch import nn
from torch.autograd import Variable
import math

class PositionalEncoding(nn.Module):
    def __init__(self, embedding_dim, max_seq_length=512, dropout=0.1):
        super(PositionalEncoding, self).__init__()
        self.embedding_dim = embedding_dim
        self.dropout = nn.Dropout(dropout)
        pe = torch.zeros(max_seq_length, embedding_dim)
        for pos in range(max_seq_length):
            for i in range(0, embedding_dim, 2):
                pe[pos, i] = math.sin(pos/(10000**(2*i/embedding_dim)))
                pe[pos, i+1] = math.cos(pos/(10000**((2*i+1)/embedding_dim)))
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x*math.sqrt(self.embedding_dim)
        seq_length = x.size(1)
        pe = Variable(self.pe[:, :seq_length],
requires_grad=False).to(x.device)
        # Add the positional encoding vector to the embedding vector
        x = x + pe
        x = self.dropout(x)
        return x
```

Add a python class to the project called MHSelfAttention with the following code in it. This code implements the multiheaded self attention using the einiops library. The mask is used during training but not in the inference phase.

```
from pickle import NONE
import numpy as np
import torch
from einops import rearrange
from torch import nn

class MHSelfAttention(nn.Module):
    def __init__(self, dim, heads=8, dim_head=None, causal=True): # e.g.,
dim=512 i.e., embedding dim
        super().__init__()
        self.dim_head = (int(dim/heads)) if dim_head is None else dim_head
        _dim = self.dim_head * heads
```

```
        self.heads = heads
        self.causal = causal
        self.to_qkv = nn.Linear(dim, _dim * 3, bias=False)
        self.W_out = nn.Linear( _dim, dim, bias=False)
        self.scale_factor = self.dim_head ** -0.5
        self.causal = causal

    def set_causal(self, causal):
        self.causal = causal

    def forward(self, x, mask=None):
        assert x.dim() == 3
        qkv = self.to_qkv(x)  # [b, n, dim*3]
        # decompose to q,k,v and cast to tuple - [3,b, heads, seq length, dim_head]
        q, k, v = tuple(rearrange(qkv, 'b n (d k h) -> k b h n d ', k=3,
h=self.heads))  # [b, heads, seq length, dim_head]

        # resulting shape will be: [batch, heads, tokens, tokens]
        scaled_dot_prod = torch.einsum('b h i d , b h j d -> b h i j', q, k) *
self.scale_factor

        #------ mask needed during training ----------------
        i = scaled_dot_prod.shape[2]
        j = scaled_dot_prod.shape[3]
        if self.causal:
            mask = torch.ones(i, j, device = 'cuda').triu_(j - i+1).bool()
        #---------------------------------------------------

        if mask is not None:
            assert mask.shape == scaled_dot_prod.shape[2:]
            scaled_dot_prod = scaled_dot_prod.masked_fill(mask, -np.inf)

        attention = torch.softmax(scaled_dot_prod, dim=-1) # attention matrix
        # output for one head
        out = torch.einsum('b h i j , b h j d -> b h i d', attention, v)

        out = rearrange(out, "b h n d -> b n (h d)")  # merge all heads into dim

        return self.W_out(out) # final linear transformation WO
```

Add a python class called TransformerBlock to the project with the following code in it. This code implements the Decoder. MHSelfAttention created earlier is one of the building blocks of this Decoder i.e., Transformer block.

```
from torch import nn

from MHSelfAttention import MHSelfAttention

class TransformerBlock(nn.Module):
   def __init__(self, dim, heads=8, dim_head=None, causal=False,
pos_embed=None, dim_linear_block=1024, dropout=0.1):
       super().__init__()
```

```
        self.mhsa = MHSelfAttention(dim=dim, heads=heads, dim_head=dim_head,
causal=causal)
        self.drop = nn.Dropout(dropout)
        self.norm_1 = nn.LayerNorm(dim)
        self.norm_2 = nn.LayerNorm(dim)

        self.linear = nn.Sequential(
            nn.Linear(dim, dim_linear_block),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(dim_linear_block, dim),
            nn.Dropout(dropout)
        )

    def set_causal(self, causal):
        self.mhsa.set_causal(causal)

    def forward(self, x, mask=None):
        y = self.norm_1(self.drop(self.mhsa(x, mask)) + x)
        return self.norm_2(self.linear(y) + y)
```

Add a python class to the project called SimpleTransformer with the following code in it. This code uses layers of transformer blocks and adds a linear layer to produce the logits at the end.

```
from torch import nn
from TransformerBlock import TransformerBlock
from PositionalEncoding import PositionalEncoding
import torch

class SimpleTransformer(nn.Module):
    def __init__(self, dim, num_unique_tokens=256, num_layers=6, heads=8,
dim_head=None, max_seq_len=1024, causal=True):
        super().__init__()
        self.max_seq_len = max_seq_len
        self.causal=causal
        self.token_emb = nn.Embedding(num_unique_tokens, dim)

        # our position embedding class
        self.pos_enc = PositionalEncoding(dim,max_seq_length=max_seq_len)

        self.block_list = [TransformerBlock(dim=dim, heads=heads,
dim_head=dim_head,causal=causal) for _ in range(num_layers)]
        self.layers = nn.ModuleList(self.block_list)

        self.to_logits = nn.Sequential(
             nn.LayerNorm(dim),
             nn.Linear(dim, num_unique_tokens)
         )

    def set_causal(self, causal):
        for b in self.block_list:
            b.set_causal(causal)
```

```
    def forward(self, x, mask=None):

        x = self.token_emb(x)
        x = x + self.pos_enc(x)
        for layer in self.layers:
            x = layer(x, mask)
        return self.to_logits(x)
```

Add a class to the project called AutoRegressiveWrapper with the following code in it. The purpose of this code is to use the Simple Transformer either in the training phase, or in the generation phase. Study the code carefully to see how the forward function is used in training and the generate function in testing or generation from a given start sequence.

```
import torch
from torch import nn
import torch.nn.functional as F

# ---------top k from logits, logits = layer before softmax
def top_k(logits, thres = 0.9):
    k = int((1 - thres) * logits.shape[-1]) # top 25
    val, ind = torch.topk(logits, k)
    probs = torch.full_like(logits, float('-inf')) # fill probs=[1,256] with -inf
    probs.scatter_(1, ind, val) # 1=dim, it will fill probs with val at ind location
    return probs                # i.e., top 25 locations will have values, rest -inf

class AutoRegressiveWrapper(nn.Module):
    def __init__(self, net, pad_value = 0):
        super().__init__()
        self.pad_value = pad_value
        self.model = net
        self.max_seq_len = net.max_seq_len

    @torch.no_grad()
    def generate(self, start_tokens, seq_len, eos_token = None, temperature = 1.,
filter_thres = 0.9):
        self.model.eval()
        device = start_tokens.device  # start tokens is the seed set of characters
        num_dims = len(start_tokens.shape) # 1, e.g., start_tokens = 1024

        if num_dims == 1:
            start_tokens = start_tokens[None, :]    # 1024 --> (1, 1024); [None, :]
is an alias for np.newaxis, it will add one more dimension

        b, t = start_tokens.shape   # b=1, e.g., t=1024. In generation, batch=1
        prev_out = start_tokens  # e.g., [1x1024]
        for _ in range(seq_len):    # seq_len = e.g., 1024
            x = prev_out[:, -self.max_seq_len:]  # x=(1, 1024); max_seq_len = 1024
            #self.model.set_causal(False) - causes generated output to be not proper
            logits = self.model(x)[:, -1, :] # x = [1x1024], logits = [1x256]

            filtered_logits = top_k(logits, thres = filter_thres)  # filtered_logits
= (1x256)
            # top 10% logits will be kept, others changed to -inf.
            probs = F.softmax(filtered_logits / temperature, dim=-1)
            predicted_char_token = torch.multinomial(probs, 1) # (1x1)
            out = torch.cat((prev_out, predicted_char_token), dim=-1)  # (1 x 1025)
```

```
            prev_out = out
            if eos_token is not None and (predicted_char_token == eos_token).all():
                break

        out = out[:, t:]   # generated output sequence after the start sequence, e.g. 1x512
        if num_dims == 1:
            out = out.squeeze(0)
        #self.model.set_causal(True) - causes generated output to be not proper
        return out


    def forward(self, x):
        xi = x[:, :-1] # x is input of size seq_len+1, :-1 will make it seq_len
        xo = x[:, 1:]  # expected output in training is shifted one to the right
        out = self.model(xi)
        logits_reorg = out.view(-1, out.size(-1))
        targets_reorg = xo.reshape(-1)
        loss = F.cross_entropy(logits_reorg, targets_reorg)
        return loss
```

Type the following code in the TransformerAMMain.py. This file sets up the Transformer for training and calls the generate function every few specified intervals.

```
import random
import tqdm
import numpy as np

import torch
import torch.optim as optim
from AutoRegressiveWrapper import AutoRegressiveWrapper
from SimpleTransformer import SimpleTransformer
import Utils
import sys
import math

# ------constants------------
NUM_BATCHES = int(1e6)
BATCH_SIZE = 16
GRADIENT_ACCUMULATE_EVERY = 1
LEARNING_RATE = 3e-4
VALIDATE_EVERY  = 500
GENERATE_EVERY  = 500
GENERATE_LENGTH = 512
SEQ_LENGTH = 1024
#--------------------------

def decode_token(token): # convert token to character
    return str(chr(max(32, token)))

def decode_tokens(tokens): # convert sequence of characters to tokens
    return ''.join(list(map(decode_token, tokens)))
```

```
def count_parameters(model): # count number of trainable parameters in the
model
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

def main():

    simple_transformer = SimpleTransformer(
        dim = 512, # embedding
        num_unique_tokens = 256,  # for character level modeling
        num_layers = 8,
        heads = 8,
        max_seq_len = SEQ_LENGTH,
        causal = True,
    )

    model = AutoRegressiveWrapper(simple_transformer)
    model.cuda()
    pcount = count_parameters(model)
    print("count of parameters in the model = ", pcount/1e6, " million")

    train_loader, val_loader, val_dataset =
Utils.get_loaders_enwiki8(SEQ_LENGTH, BATCH_SIZE)
    optim = torch.optim.Adam(model.parameters(), lr = LEARNING_RATE)      #
optimizer

    # --------training---------
    for i in tqdm.tqdm(range(NUM_BATCHES), mininterval = 10., desc =
'training'):
        model.train()
        total_loss = 0
        for __ in range(GRADIENT_ACCUMULATE_EVERY):
            loss = model(next(train_loader))
            loss.backward()
        if (i%100 == 0):
            print(f'training loss: {loss.item()} -- iteration = {i}')

        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        optim.step()
        optim.zero_grad()

        if i % VALIDATE_EVERY == 0:
            model.eval()
            total_len2 = 0
            total_loss2 = 0
            val_count = 1000  # number of validations to compute average BPC
            with torch.no_grad():
                for v in range(0,val_count):
                    loss = model(next(val_loader))
                    total_loss += loss.item()
                    loss_m = loss.mean()
                    total_loss2 += SEQ_LENGTH * loss.float().item() #seq_len
                    total_len2 += SEQ_LENGTH
                print(f'----------validation loss: {total_loss/val_count}')
                print(f'Perplexity : {math.exp(total_loss/val_count)}, BPC:
{total_loss/val_count*np.log2(2.7173)}')
```

```
                bpc2 = (total_loss2/total_len2)/math.log(2)
                print("BPC 2 = ", bpc2)
                total_loss = 0
        if i % GENERATE_EVERY == 0:
            model.eval()
            inp = random.choice(val_dataset)[:-1]
            input_start_sequence = decode_tokens(inp)
            print("----------start input----------")
            print(f'%s \n\n', (input_start_sequence))
            print("----------end of start input----------")
            sample = model.generate(inp, GENERATE_LENGTH)
            output_str = decode_tokens(sample)
            print("----------generated output----------")
            print(output_str)
            print("----------end generated output----------")

if __name__ == "__main__":
    sys.exit(int(main() or 0))
```

Build and test the above code. It may take over a day to train it to a semi decent level where the BPC drops to around 1.3.
After about 10 hours of training on an RTX 3090/4090 GPU, the BPC, training loss, and generated output appear as: