Assignment 3: Simple Transformer Implementation

Prepared by: Pranjal Raj Ghimire
StudentID: 1241252

In this assignment, We implemented a simple transformer model from scratch. A character level language model was built based on the Decoder style architecture that only focuses on generation of one output at a time.

The dataset used for this is the Enwik8 dataset. It has 100 million characters, out of which 90 million will be used for training and the remaining 10 million will be split equally into test and validation sets.

Firstly, MyNLPDataset.py is initialized with the following code:

Assignment 3 > MyNLPDataSet.py > MyNLPDataset > __len__

```python
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
```

This is used to load the data. The __getitem __ method is used to get the sequence length number of characters from a random position in the dataset. We randomly need to feed in the data during training, and this method helps to bring in this stochastic behavior when training the model.

Next, a Utils.py file is created with the following code:

Assignment 3 > Utils.py > get_loaders_enwiki8

```python
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
```

The get_loaders_enwiki8 is the method that reads and unzips the downloaded enwik8.gz file. After that, it reads from the data and creates the training and validation datasets. It ultimately returns the training dataloader, validation dataloader and the validation dataset.

Assignment 3 > PositionalEncoding.py > PositionalEncoding > forward

```python
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
```

A class PositionalEncoding is created to implement the positional encoding part of the Transformer. Positional Encoding is used by the model to understand the sequential order of the tokens.This enables the model to understand the relationships based on the positions within the input sequence. Sine and Cosine functions are used to find the position encoding for the different tokens. This allows the model to learn relationships between positions. The distance between two positions can be captured through the relative values of their encodings. Also, these functions provide smooth variations in the encoding. When combined with the self attention mechanism, positional encodings help to ensure that the model considers the sequence's structure when working with different tokens.

Next, the MHSelfAttention class is created. It implements the multi headed self attention using einops. Upper triangular masking is used during training but not during testing.

Assignment 3 > MHSelfAttention.py > MHSelfAttention > forward

```python
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

    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)
```

The code above does the MultiHeaded self attention.

The self.to_qkv is a linear layer that transforms the input embeddings into Query, Key and Value representations. The output dimension is three times the dimensionality of the heads, allowing to separate the Q, K and V matrices.

If the causal attention is enabled, a triangular mask is initialized to prevent attending to the future tokens in the sequence during training. Then, it is applied to the attention scores, replacing the masked positions (upper triangle) with negative infinity to ensure these positions have zero probability after softmax is applied to it.

Then, they are passed through the softmax function to get the attention weights.

Finally, the output calculation is done in the two out variables. The out = A * V is calculated, and the output from all heads are concatenated. The combined output is passed through the W_out linear layer to project it back to its original embedding dimension to be sent into another layer.

Then, a python class TransformerBlock is added. The code implements the decoder mechanism. The MHSelfAttention class created just before this is the building block of this Decoder or Transformer block. The code for the transformer block is given below:

Assignment 3 > TransformerBlock.py > ...

```python
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)
```

The transformerBlock initializes the MHSelfAttention class; which is the multi head self attention layer created in the code snippet just before this page. Here, we can see that a dropout layer is initialized for regularization. Two Layer normalization layers - LayerNorm are also initialized, which helps to improve the training by normalizing the inputs across the different layers.
The Forward method computes the forward propagation of the transformer block. The y variable computes the multi headed self attention applied to the input x and adds the original input (x). It applies dropout and normalizes the result using norm_1. Then, the output is again passed through the linear layer, added with the original value of y, and normalized using norm_2 and the final output is returned.

A SimpleTransformer Class is created with the following code:

Assignment 3 > SimpleTransformer.py > SimpleTransformer > forward

```python
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)
```

This code implements a basic transformer model embedding layers, positional encoding, multiple transformer blocks (block_list), and a linear layer to output the predictions.

Next, an AutoRegressiveWrapper class is added with the following code:

Assignment 3 > AutoRegressiveWrapper.py > AutoRegressiveWrapper > generate

```python

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=1.0, 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)  # e.g., start_tokens = 1024
        if num_dims == 1:
            start_tokens = start_tokens[None, :]  # (1, 1024)

        b, t = start_tokens.shape  # b=1, e.g., t=1024
        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)
            logits = self.model(x)[:, -1, :]  # 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
        if num_dims == 1:
            out = out.squeeze(0)

        return out

    def forward(self, x):
        xi = x[:, :-1]  # input of size seq_len+1
        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
```

The forward code is used for the training phase of the model. It computes the loss, based on the input tokens and expected outputs During training, we have a loop where batches of data are fed into the model. The input for each batch is expected to have one extra token, i.e. shifted one position to the left by one position.

The generate method is used for testing or inference, for generating sequences based on the starting context. This method allows the model to produce new tokens iteratively (sampling from the probability distribution of the next token. Typically, we provide the model with a sequence of starting tokens and specify how many tokens we want it to generate. The generate method performs the generation by repeatedly calling the model with the current sequence, sampling the next token and appending it to the next sequence.

Together, these methods of AutoRegressiveWrapper enable the model to learn from the data during training and generate sequences during testing.

Finally, all the code above is set up to create a Transformer for training and calls the generate function every few intervals. The code is given by the following snippet:

Assignment 3 > transformerPGmain.py > main

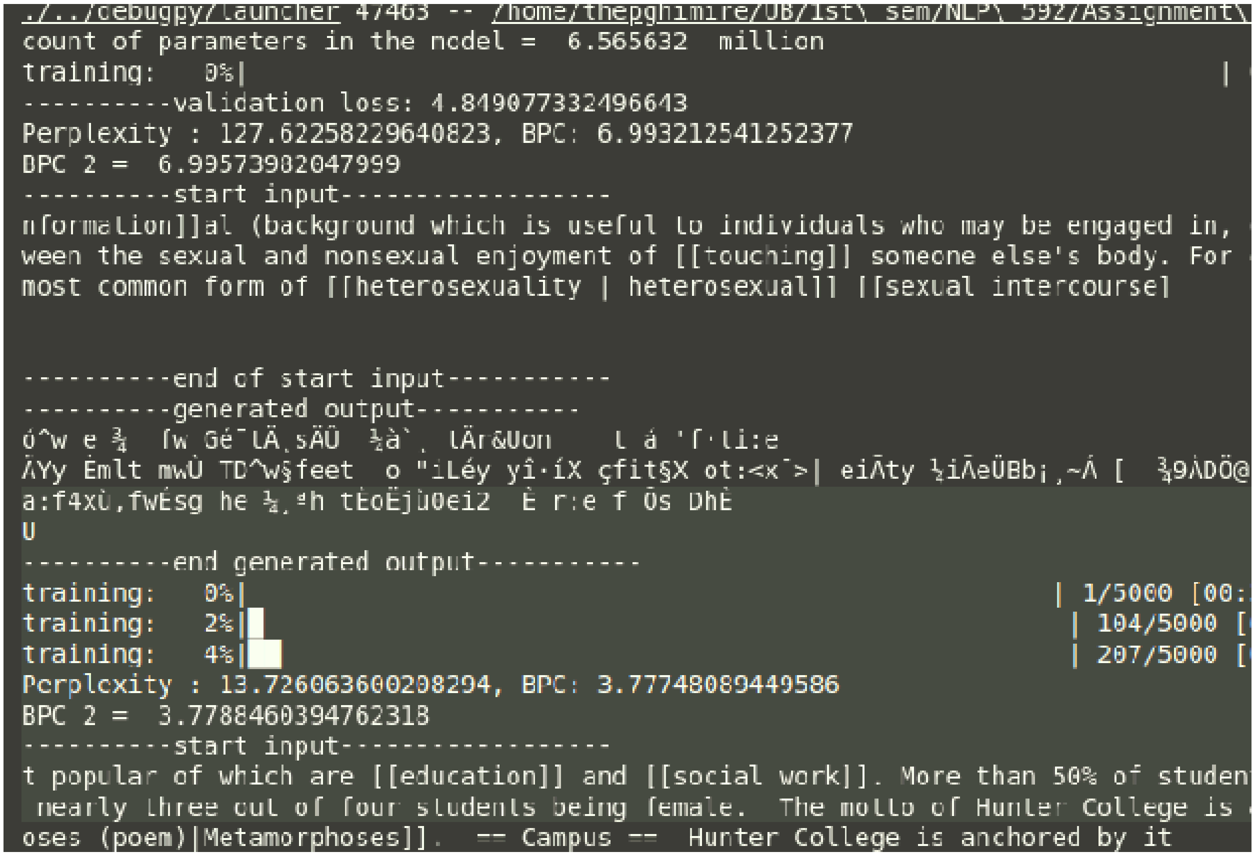
```python
# ------constants------------
NUM_BATCHES = int(5e3)
BATCH_SIZE = 8
GRADIENT_ACCUMULATE_EVERY = 1
LEARNING_RATE = 3e-4
VALIDATE_EVERY = 240
GENERATE_EVERY = 240
GENERATE_LENGTH = 256
SEQ_LENGTH = 512
#---------------------------

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():
    #actual
    # 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,
    # )
    #modified
    simple_transformer = SimpleTransformer(
        dim=512,  # embedding
        num_unique_tokens=256,  # for character level modeling
        num_layers=3,
        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") #####1e6

    train_loader, val_loader, val_dataset = Utils.get_loaders_enwik8(SEQ_LENGTH, BATCH_SIZE)
```

Assignment 3 > transformerPGmain.py > main

```python
def main():
    train_loader, val_loader, val_dataset = Utils.get_loaders_enwiki8(SEQ_LENGTH, BATCH_SIZE)
    optim = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE)  # optim

    # --------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(val_count):
                    loss = model(next(val_loader))
                    total_loss += loss.item()
                    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
                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'{input_start_sequence}\n\n')
            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----------")
```

I have changed the batch_size, layers, and a few other constants for it to run on my system. When I had the initial values, it returned a memory error.

The final piece of this code implements the training and evaluation of the simple transformer model for character level Language modeling tasks. The constants and other hyperparameters are initialized, and the training loop is ran. Then, there is a separate validation and text generation part where the model evaluates its performance on the validation after a certain “VALID_EVERY” iterations, which in my code is set to 240 iterations. The model also generates every 240 iterations as setup on the constants. The output received on my system (just made sure that the training loop ran with a reduced batch_size and layers) was as follows:


```
./../debugpy/launcher 47463 -- /home/thepghimire/UB/1st\ sem/NLP\ 592/Assignment\
count of parameters in the model =  6.565632  million
training:   0%|                                                              |
----------validation loss: 4.849077332496643
Perplexity : 127.62258229640823, BPC: 6.993212541252377
BPC 2 =  6.99573982047999
----------start input------------------
nformation]]al (background which is useful to individuals who may be engaged in,
ween the sexual and nonsexual enjoyment of [[touching]] someone else's body. For
most common form of [[heterosexuality | heterosexual]] [[sexual intercourse]


----------end of start input-----------
----------generated output-----------
ó^w e ¾  ſw Gé¯LÄ¸sÄÜ  ½à`¸ LÄr&Uon    L á 'ſ·Li:e
ΛYy Èmlt mwÙ TD^w§feet  o "iLéy yî·íX çfit§X ot:<x¯>| eiΛty ½iΛeÜBb¡¸~Λ [  ¾9ΛDÖ@
a:f4xÙ,fwÉsg he ¾¸ªh tÈoËjÙ0ei2  È r:e f Ös DhÈ
U
----------end generated output-----------
training:   0%|                                                 | 1/5000 [00:
training:   2%|█                                                 | 104/5000 [
training:   4%|██                                                | 207/5000 [
Perplexity : 13.726063600208294, BPC: 3.77748089449586
BPC 2 =  3.7788460394762318
----------start input------------------
t popular of which are [[education]] and [[social work]]. More than 50% of studen
 nearly three out of four students being female.  The motto of Hunter College is
oses (poem)|Metamorphoses]].  == Campus ==  Hunter College is anchored by it
```
Since the full picture was difficult to read, this picture is a cropped part of the OUTPUT.png picture. The full picture can be found in the unzipped assignment folder.