Code fro E4.4

```
# -*- coding: utf-8 -*-
"""Copy of A2Exercise3.ipynb

Automatically generated by Colaboratory.

Original file is located at
   https://colab.research.google.com/drive/1ttGr0qC5qRdyyoVtclaRoaOXIMdfKT1j
"""

# Make a copy of this notebook for your submission and fill in your details here
# Name: Yijie (Jackie) Zhu
# Waterloo ID: 20832931

#Run this chunk of code:

# init libraries
import torch
import torch.nn as nn
import numpy as np
from torch.nn import functional as F
import math

# hyperparameters
batch_size = 16
block_size = 256
max_iters = 50
eval_interval = 500
learning_rate = 3e-4
device = 'cuda' if torch.cuda.is_available() else 'cpu'
eval_iters = 200
n_embd = 384
n_head = 6
n_layer = 6
dropout = 0.2

torch.manual_seed(123123)

# Here we process the .txt file and create our training and validation set

############### ADD NEW FILE DIR #######
with open('sample_data/frankenstein.txt', 'r', encoding='utf-8') as f:
```

```
   text = f.read()

# Part IV. (a)
#### FILL IN THE EMPTY CODE HERE #### (2 marks)
# For this part, reuse your code from Exercise 4 Part 1.(a)

chars = list(dict.fromkeys(sorted(text.split(" "))))
vocab_size = len(chars)


stoi = {} ## create a mapping from characters to integers
itos = {} ## create a mapping from integers to characters

# filling mappings
for i, s in enumerate(chars):
       itos[i] = s
       stoi[s] = i

## write a function that takes a string and returns a list of integers
def encode(s):
 i_list = []
 for word in s.split(" "):
   i_list.append(stoi[word])
 return i_list

## write a function that takes a list of integers and returns a string
def decode(i):
 s_list = []
 for id in i:
   s_list.append(itos[id])
 return ' '.join(s_list)

# Now we split our data into a 80/20 train and val splits
data = torch.tensor(encode(text), dtype=torch.long)
n = int(0.8*len(data))
train_data = data[:n]
val_data = data[n:]

# We first create an object for computing the self-attention for a single attention
head and then we compute the multi-headed attention using this object.

# Part IV. (b)
```

```
#### FILL IN THE EMPTY CODE HERE #### (6 marks)
# For this part you can reuse your code from Exercise 4 Part 1.(b) and (c)
class Head(nn.Module):
   def __init__(self, head_size):
       super().__init__()
       self.key = nn.Linear(n_embd, head_size, bias=False)
       self.query = nn.Linear(n_embd, head_size, bias=False)
       self.value = nn.Linear(n_embd, head_size, bias=False)
       self.register_buffer('tril', torch.tril(torch.ones(block_size, block_size)))
       self.dropout = nn.Dropout(dropout)

   def forward(self, x):
       B, T, C = x.shape
       ## create a self-attention mechanism that outputs a context vector for an input
x

       k = self.key(x)
       q = self.query(x)
       v = self.value(x)

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

       wx = wx.masked_fill(self.tril[:T, :T] == 0, float('-inf')) # (B, T, T)
       alphaxi = F.softmax(wx / math.sqrt(len(k)), dim=-1)
       alphaxi = self.dropout(alphaxi)
       c = torch.matmul(alphaxi, v)

       C = c.to(device)

       # your output should be of size (batch, time-step, head size)
       return C

class MultiHeadAttention(nn.Module):
   def __init__(self, num_heads, head_size):
       super().__init__()
       self.heads = nn.ModuleList([Head(head_size) for _ in range(num_heads)])
       self.proj = nn.Linear(head_size * num_heads, n_embd)
       self.dropout = nn.Dropout(dropout)

   def forward(self, x):
       context_list = [head.forward(x) for head in self.heads]
       out = torch.cat(context_list, dim=-1)
       out = self.dropout(self.proj(out))  # here we apply dropout on a linear layer
```

```python
        return out

# Now we create  our Feed Forward Network and Attention Block

class FeedFoward(nn.Module):
    """ Here we define a feed forward network with the following structure """
    # for n_embd: embedding dimension
    # a linear layer (n_embd x 4 * n_embd)
    # a ReLU layer
    # another linear layer (n_embd x 4 * n_embd)
    # a Dropout layer

    def __init__(self, n_embd):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd),
            nn.Dropout(dropout)
        )

    def forward(self, x):
        return self.net(x)

class AttentionBlock(nn.Module):
    """ Here we create our Attention Block with the following structure """
    # for n_embd: embedding dim, n_head: number of heads
    # layer norm layer
    # multi-head attention layer
    # layer norm layer
    # feed forward layer

    def __init__(self, n_embd, n_head):
        super().__init__()
        head_size = n_embd // n_head
        self.sa = MultiHeadAttention(n_head, head_size)
        self.ffwd = FeedFoward(n_embd)
        self.ln1 = nn.LayerNorm(n_embd)
        self.ln2 = nn.LayerNorm(n_embd)

    def forward(self, x):
        x = x + self.sa(self.ln1(x))
```

```
        x = x + self.ffwd(self.ln2(x))
        return x

# Now we put everything together for our final model

class ECE457BGPT(nn.Module):
    # a token embedding layer of shape: vocab_size x embedding_dim
    # a position embedding layer of shape: block_size x embedding_dim
    # a sequential multi_block layer that creates n AttentionBlocks, each of shape:
n_embd x n_head
    # a LayerNorm layer of size n_embd
    # a final Linear layer of shape: n_embd x vocab_size
    def __init__(self):
        super().__init__()
        ## create the layers that the model will use given the structure above
        self.token_embedding_table = nn.Embedding(vocab_size, n_embd)
        self.position_embedding_table = nn.Embedding(block_size, n_embd)
        self.blocks = nn.Sequential(*[AttentionBlock(n_embd, n_head=n_head) for _ in
range(n_layer)])
        self.ln_f = nn.LayerNorm(n_embd) # final layer norm
        self.lm_head = nn.Linear(n_embd, vocab_size)

        self.apply(self._init_weights) ## this will apply the function below to
initialize the weights and to improve the model's training speed

    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)

    def forward(self, idx, targets=None):
        B, T = idx.shape
        # idx and targets are both (B,T) tensor of integers
        tok_emb = self.token_embedding_table(idx) # (B,T,C)
        pos_emb = self.position_embedding_table(torch.arange(T, device=device)) # (T,C)
        x = tok_emb + pos_emb # (B,T,C)
        x = self.blocks(x) # (B,T,C)
        x = self.ln_f(x) # (B,T,C)
        logits = self.lm_head(x) # (B,T,vocab_size)
```

```
        if targets is None:
            loss = None
        else:
            ## compute the cross-entropy loss for the logits and the target values
            B, T, C = logits.shape
            logits = logits.view(B*T, C)
            targets = targets.view(B*T)
            loss = F.cross_entropy(logits, targets)

        return logits, loss

    def generate(self, idx, max_new_tokens):
        for _ in range(max_new_tokens):
            idx_cond = idx[:, -block_size:] ## crop idx to the last block_size tokens
            logits, loss = self(idx_cond) ## get the predictions
            logits = logits[:, -1, :]
            probs = F.softmax(logits, dim=-1) ## apply softmax to get probabilities
            idx_next = torch.multinomial(probs, num_samples=1)
            idx = torch.cat((idx, idx_next), dim=1)
        return idx

# Now we optimize our model and evaluate it's performance
model = ECE457BGPT()
m = model.to(device)
optimizer =  torch.optim.AdamW(model.parameters(), lr=learning_rate) ## PyTorch AdamW
optimizer with model params and the global learning rate defined at the start of this
notebook.

# helper function that creates a small batch of the data to validate the models
performance
def get_batch(split):
    data = train_data if split == 'train' else val_data
    ix = torch.randint(len(data) - block_size, (batch_size,))
    x = torch.stack([data[i:i+block_size] for i in ix])
    y = torch.stack([data[i+1:i+block_size+1] for i in ix])
    x, y = x.to(device), y.to(device)
    return x, y


## for loop for estimating the train and val loss.
```

```python
# prints the average train and val losses for the previous steps at every timestep of
500
# Uses the helper function to sample a batch of training data, evaluates the loss on
it and uses the optimizer to backward prop through the model

@torch.no_grad()
def estimate_loss():
   out = {}
   model.eval()
   for split in ['train', 'val']:
       losses = torch.zeros(eval_iters)
       for k in range(eval_iters):
           X, Y = get_batch(split)
           logits, loss = model(X, Y)
           losses[k] = loss.item()
       out[split] = losses.mean()
   model.train()
   return out

for iter in range(max_iters):
 ## Your code here
   print("Iteration: " + str(iter))
  # every once in a while evaluate the loss on train and val sets
   if iter % eval_interval == 0 or iter == max_iters - 1:
       losses = estimate_loss()
       print(f"step {iter}: train loss {losses['train']:.4f}, val loss
{losses['val']:.4f}")

   # sample a batch of data
   xb, yb = get_batch('train')

   # evaluate the loss
   logits, loss = model(xb, yb)
   optimizer.zero_grad(set_to_none=True)
   loss.backward()
   optimizer.step()

 # pass ## remove this when you write your code

#You can finally test your model's ability to generate text using this line of code!
context = torch.zeros((1, 1), dtype=torch.long, device=device)
print(decode(m.generate(context, max_new_tokens=500)[0].tolist()))
```