CSCI 4140: Natural Language Processing

CSCI/DASC 6040: Computational Analysis of Natural Languages

Spring 2024

Homework 5 - Building a GPT

Due Sunday, March 24, at 11:59 PM

Adapted from the companion notebook to the Zero To Hero video on GPT.

1. Select a dataset

Andrej Karpathy used a subset of Shakespeare's works in his nanoGPT. **Pick a different author to train this GPT.** For example, Project Gutenberg is a library of over 70,000 free eBooks. To download The Complete Works of William Shakespeare by William Shakespeare in text format, you can download the Plain Text UTF-8 file.

```
# We always start with a dataset to train on. Let's download the tiny shakespeare dataset
!wget https://www.gutenberg.org/cache/epub/100/pg100.txt
# Replace "https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt" with the URL for your author
# Save the downloaded file as "input.txt"
```

```
--2024-03-21 18:41:53--  https://www.gutenberg.org/cache/epub/100/pg100.txt
Resolving www.gutenberg.org (www.gutenberg.org)... 152.19.134.47, 2610:28:3090:3000:0:bad:cafe:47
Connecting to www.gutenberg.org (www.gutenberg.org)|152.19.134.47|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 5638549 (5.4M) [text/plain]
Saving to: ‘pg100.txt.5’

pg100.txt.5         100%[===================>]   5.38M  34.7MB/s    in 0.2s

2024-03-21 18:41:54 (34.7 MB/s) - ‘pg100.txt.5’ saved [5638549/5638549]
```

```
# read it in to inspect it
with open('pg100.txt.5', 'r', encoding='utf-8') as f:
    text = f.read()
```

Get the length of this dataset (in characters)

```
print("length of dataset in characters: ", len(text))
```

```
length of dataset in characters:  5378693
```

```
# let's look at the first 1000 characters
print(text[:1000])
```

```
The Project Gutenberg eBook of The Complete Works of William Shakespeare

This ebook is for the use of anyone anywhere in the United States and
most other parts of the world at no cost and with almost no restrictions
whatsoever. You may copy it, give it away or re-use it under the terms
of the Project Gutenberg License included with this ebook or online
at www.gutenberg.org. If you are not located in the United States,
you will have to check the laws of the country where you are located
before using this eBook.

Title: The Complete Works of William Shakespeare


Author: William Shakespeare

Release date: January 1, 1994 [eBook #100]
                Most recently updated: January 18, 2024

Language: English
```

```
*** START OF THE PROJECT GUTENBERG EBOOK THE COMPLETE WORKS OF WILLIAM SHAKESPEARE ***
The Complete Works of William Shakespeare

by William Shakespeare




                    Contents

    THE SONNETS
    ALL’S WELL THAT ENDS WELL
    THE TRAGEDY OF ANTONY AND CLEOPATRA
```

Get the vocabulary for this dataset, and its length

```
# here are all the unique characters that occur in this text
chars = sorted(list(set(text)))
vocab_size = len(chars)
print(''.join(chars))
print(vocab_size)
```

```
 !#$%&'()*,-./0123456789:;?ABCDEFGHIJKLMNOPQRSTUVWXYZ[]_abcdefghijklmnopqrstuvwxyzÀÆÇÉàâæçèéêëîœ—‘’“”•…™
107
```

Create the encoder (string -> list of integers) and decoder (list of integers -> string)

```
# create a mapping from characters to integers
stoi = { ch:i for i,ch in enumerate(chars) }
itos = { i:ch for i,ch in enumerate(chars) }
encode = lambda s: [stoi[c] for c in s] # encoder: take a string, output a list of integers
decode = lambda l: ''.join([itos[i] for i in l]) # decoder: take a list of integers, output a string

print(encode("hii there"))
print(decode(encode("hii there")))
```

```
[65, 66, 66, 2, 77, 65, 62, 75, 62]
hii there
```

Load encoded data into a tensor

```
# let's now encode the entire text dataset and store it into a torch.Tensor
import torch # we use PyTorch: https://pytorch.org
data = torch.tensor(encode(text), dtype=torch.long)
print(data.shape, data.dtype)
print(data[:1000]) # the 1000 characters we looked at earier will to the GPT look like this
```

```
        77,  65,   2,  58,  69,  70,  72,  76,  77,   2,  71,  72,   2,  75,
        62,  76,  77,  75,  66,  60,  77,  66,  72,  71,  76,   1,  80,  65,
        58,  77,  76,  72,  62,  79,  62,  75,  14,   2,  53,  72,  78,   2,
        70,  58,  82,   2,  60,  72,  73,  82,   2,  66,  77,  12,   2,  64,
        66,  79,  62,   2,  66,  77,   2,  58,  80,  58,  82,   2,  72,  75,
         2,  75,  62,  13,  78,  76,  62,   2,  66,  77,   2,  78,  71,  61,
        62,  75,   2,  77,  65,  62,   2,  77,  62,  75,  70,  76,   1,  72,
        78,  76,  66,  71,  64,   2,  77,  65,  66,  76,   2,  62,  30,  72,
        72,  68,  14,   1,   1,  48,  66,  77,  69,  62,  26,   2,  48,  65,
        62,   2,  31,  72,  70,  73,  69,  62,  77,  62,   2,  51,  72,  75,
        68,  76,   2,  72,  63,   2,  51,  66,  69,  69,  66,  58,  70,   2,
        47,  65,  58,  68,  62,  76,  73,  62,  58,  75,  62,   1,   1,   1,
        29,  78,  77,  65,  72,  75,  26,   2,  51,  66,  69,  69,  66,  58,
        70,   2,  47,  65,  58,  68,  62,  76,  73,  62,  58,  75,  62,   1,
         1,  46,  62,  69,  62,  58,  76,  62,   2,  61,  58,  77,  62,  26,
         2,  38,  58,  71,  78,  58,  75,  82,   2,  17,  12,   2,  17,  25,
        25,  20,   2,  55,  62,  30,  72,  72,  68,   2,   4,  17,  16,  16,
        56,   1,   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,
         2,   2,   2,   2,  41,  72,  76,  77,   2,  75,  62,  60,  62,  71,
        77,  69,  82,   2,  78,  73,  61,  58,  77,  62,  61,  26,   2,  38,
        58,  71,  78,  58,  75,  82,   2,  17,  24,  12,   2,  18,  16,  18,
        20,   1,   1,  40,  58,  71,  64,  78,  58,  64,  62,  26,   2,  33,
        71,  64,  69,  66,  76,  65,   1,   1,   1,   1,  11,  11,  11,   2,
        47,  48,  29,  46,  48,   2,  43,  34,   2,  48,  36,  33,   2,  44,
        46,  43,  38,  33,  31,  48,   2,  35,  49,  48,  33,  42,  30,  33,
        46,  35,   2,  33,  30,  43,  43,  39,   2,  48,  36,  33,   2,  31,
        43,  41,  44,  40,  33,  48,  33,   2,  51,  43,  46,  39,  47,   2,
        43,  34,   2,  51,  37,  40,  40,  37,  29,  41,   2,  47,  36,  29,
        39,  33,  47,  44,  33,  29,  46,  33,   2,  11,  11,  11,   1, 106,
        48,  65,  62,   2,  31,  72,  70,  73,  69,  62,  77,  62,   2,  51,
        72,  75,  68,  76,   2,  72,  63,   2,  51,  66,  69,  69,  66,  58,
        70,   2,  47,  65,  58,  68,  62,  76,  73,  62,  58,  75,  62,   1,
         1,  59,  82,   2,  51,  66,  69,  69,  66,  58,  70,   2,  47,  65,
        58,  68,  62,  76,  73,  62,  58,  75,  62,   1,   1,   1,   1,   1,
         2,   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,   2,
         2,   2,   2,   2,   2,   2,  31,  72,  71,  77,  62,  71,  77,  76,
         1,   1,   2,   2,   2,   2,  48,  36,  33,   2,  47,  43,  42,  42,
        33,  48,  47,   1,   2,   2,   2,   2,  29,  40,  40, 100,  47,   2,
        51,  33,  40,  40,   2,  48,  36,  29,  48,   2,  33,  42,  32,  47,
         2,  51,  33,  40,  40,   1,   2,   2,   2,   2,  48,  36,  33,   2,
        48,  46,  29,  35,  33,  32,  53,   2,  43,  34,   2,  29,  42,  48,
        43,  42,  53,   2,  29,  42,  32,   2,  31,  40,  33,  43,  44,  29,
        48,  46,  29,   1,   2,   2])
```

Split the data into training and validation sets

```
# Let's now split up the data into train and validation sets
n = int(0.9*len(data)) # first 90% will be train, rest val
train_data = data[:n]
val_data = data[n:]
```

```
block_size = 8
train_data[:block_size+1]
```

```
tensor([106,  48,  65,  62,   2,  44,  75,  72,  67])
```

```
x = train_data[:block_size]
y = train_data[1:block_size+1]
for t in range(block_size):
    context = x[:t+1]
    target = y[t]
    print(f"when input is {context} the target: {target}")
```

```
when input is tensor([106]) the target: 48
when input is tensor([106,  48]) the target: 65
when input is tensor([106,  48,  65]) the target: 62
when input is tensor([106,  48,  65,  62]) the target: 2
when input is tensor([106,  48,  65,  62,   2]) the target: 44
when input is tensor([106,  48,  65,  62,   2,  44]) the target: 75
when input is tensor([106,  48,  65,  62,   2,  44,  75]) the target: 72
when input is tensor([106,  48,  65,  62,   2,  44,  75,  72]) the target: 67
```

```
torch.manual_seed(1337)
batch_size = 4 # how many independent sequences will we process in parallel?
block_size = 8 # what is the maximum context length for predictions?

def get_batch(split):
    # generate a small batch of data of inputs x and targets y
    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])
    return x, y

xb, yb = get_batch('train')
print('inputs:')
print(xb.shape)
print(xb)
print('targets:')
print(yb.shape)
print(yb)

print('----')

for b in range(batch_size): # batch dimension
    for t in range(block_size): # time dimension
        context = xb[b, :t+1]
        target = yb[b,t]
        print(f"when input is {context.tolist()} the target: {target}")
```

```
inputs:
torch.Size([4, 8])
tensor([[71, 77, 72, 71, 82,  2, 60, 58],
        [77, 65, 82,  2, 79, 58, 69, 72],
        [69, 72, 75, 61, 12,  2, 70, 82],
        [ 2, 73, 75, 58, 82,  2, 82, 72]])
targets:
torch.Size([4, 8])
tensor([[77, 72, 71, 82,  2, 60, 58, 71],
        [65, 82,  2, 79, 58, 69, 72, 78],
        [72, 75, 61, 12,  2, 70, 82,  2],
        [73, 75, 58, 82,  2, 82, 72, 78]])
----
when input is [71] the target: 77
when input is [71, 77] the target: 72
when input is [71, 77, 72] the target: 71
when input is [71, 77, 72, 71] the target: 82
when input is [71, 77, 72, 71, 82] the target: 2
when input is [71, 77, 72, 71, 82, 2] the target: 60
when input is [71, 77, 72, 71, 82, 2, 60] the target: 58
when input is [71, 77, 72, 71, 82, 2, 60, 58] the target: 71
when input is [77] the target: 65
when input is [77, 65] the target: 82
when input is [77, 65, 82] the target: 2
when input is [77, 65, 82, 2] the target: 79
when input is [77, 65, 82, 2, 79] the target: 58
when input is [77, 65, 82, 2, 79, 58] the target: 69
when input is [77, 65, 82, 2, 79, 58, 69] the target: 72
when input is [77, 65, 82, 2, 79, 58, 69, 72] the target: 78
when input is [69] the target: 72
when input is [69, 72] the target: 75
when input is [69, 72, 75] the target: 61
when input is [69, 72, 75, 61] the target: 12
when input is [69, 72, 75, 61, 12] the target: 2
when input is [69, 72, 75, 61, 12, 2] the target: 70
when input is [69, 72, 75, 61, 12, 2, 70] the target: 82
when input is [69, 72, 75, 61, 12, 2, 70, 82] the target: 2
when input is [2] the target: 73
when input is [2, 73] the target: 75
when input is [2, 73, 75] the target: 58
when input is [2, 73, 75, 58] the target: 82
when input is [2, 73, 75, 58, 82] the target: 2
when input is [2, 73, 75, 58, 82, 2] the target: 82
when input is [2, 73, 75, 58, 82, 2, 82] the target: 72
when input is [2, 73, 75, 58, 82, 2, 82, 72] the target: 78
```

```
print(xb) # our input to the transformer
```

```
tensor([[71, 77, 72, 71, 82,  2, 60, 58],
        [77, 65, 82,  2, 79, 58, 69, 72],
        [69, 72, 75, 61, 12,  2, 70, 82],
        [ 2, 73, 75, 58, 82,  2, 82, 72]])
```

2. Bigram language model

```
import torch
import torch.nn as nn
from torch.nn import functional as F
torch.manual_seed(1337)

class BigramLanguageModel(nn.Module):

    def __init__(self, vocab_size):
        super().__init__()
        # each token directly reads off the logits for the next token from a lookup table
        self.token_embedding_table = nn.Embedding(vocab_size, vocab_size)

    def forward(self, idx, targets=None):

        # idx and targets are both (B,T) tensor of integers
        logits = self.token_embedding_table(idx) # (B,T,C)

        if targets is None:
            loss = None
        else:
            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):
        # idx is (B, T) array of indices in the current context
        for _ in range(max_new_tokens):
            # get the predictions
            logits, loss = self(idx)
            # focus only on the last time step
            logits = logits[:, -1, :] # becomes (B, C)
            # apply softmax to get probabilities
            probs = F.softmax(logits, dim=-1) # (B, C)
            # sample from the distribution
            idx_next = torch.multinomial(probs, num_samples=1) # (B, 1)
            # append sampled index to the running sequence
            idx = torch.cat((idx, idx_next), dim=1) # (B, T+1)
        return idx

m = BigramLanguageModel(vocab_size)
logits, loss = m(xb, yb)
print(logits.shape)
print(loss)

print(decode(m.generate(idx = torch.zeros((1, 1), dtype=torch.long), max_new_tokens=100)[0].tolist()))
```

```
torch.Size([32, 107])
tensor(5.4946, grad_fn=<NllLossBackward0>)
        ë !(xœ“I$B”è'LOoyNx_eDY:a$%     BFx3)PQ’*âÇxwëJÉ'odwcYDe…àOAvBÆJ,‘Hë™lB57s0j” ( âeDeH‘TF—am_jEK;El;h'$…o
```

Use the AdamW optimizer

```
# create a PyTorch optimizer
optimizer = torch.optim.AdamW(m.parameters(), lr=1e-3)
```

Questions

1. (10 points) How many steps did you train the model for?
2. (15 points) How does the value of the loss function change with the number of steps? Provide a few examples, such as, after 100 steps the loss was 4.656, and after 1,000 steps the loss was 4.231.

1. General steps until this point: 8(Select a dataset, get length of dataset,get vocab,create encoder, load encoded data, split data into training and validation sets, bigram language model, then adamw optimizer). Coding steps: 14 cells were run until this point
2. as the steps increase the loss decreases. At 100 the loss is around 5, after 1000 the loss is 3.89, after 10000 the loss is 2.45

```
batch_size = 32
for steps in range(10000): # increase number of steps for good results...

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

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

print(loss.item())
```

```
2.410722017288208
```

```
print(decode(m.generate(idx = torch.zeros((1, 1), dtype=torch.long), max_new_tokens=500)[0].tolist()))
```

```
        aweact, ing, he sofind’d ng. ame mantusthay whe,
Ancand; t nghere, myom

Peet, litan here mug t tode? owhindonowie h, o’wave d ssbe miom HI, batwncheal tin, thay wieatrad misid to himot wome m.
Whed h fofe s ‘ANo Hin, ouf.
SCaind me ar knevebaghed che m w belo

WBy de’sess foo LEn-v’trt trchan ff Kn, rede melomandllllsthe mpust Gopit our meng r  w thanghy ncef.
Of bo, be f, warve we aril hicounsiend isthewee s mo Se n, oupeave
S.

AR ICBESot plvenesee O.
F.
Tourc dr pofousherno


Ath atenan_RUSi
```

Optional: The mathematical trick in self-attention

```
# toy example illustrating how matrix multiplication can be used for a "weighted aggregation"
torch.manual_seed(42)
a = torch.tril(torch.ones(3, 3))
a = a / torch.sum(a, 1, keepdim=True)
b = torch.randint(0,10,(3,2)).float()
c = a @ b
print('a=')
print(a)
print('--')
print('b=')
print(b)
print('--')
print('c=')
print(c)
```

```
a=
tensor([[1.0000, 0.0000, 0.0000],
        [0.5000, 0.5000, 0.0000],
        [0.3333, 0.3333, 0.3333]])
--
b=
tensor([[2., 7.],
        [6., 4.],
        [6., 5.]])
--
c=
tensor([[2.0000, 7.0000],
        [4.0000, 5.5000],
        [4.6667, 5.3333]])
```

```
# consider the following toy example:

torch.manual_seed(1337)
B,T,C = 4,8,2 # batch, time, channels
x = torch.randn(B,T,C)
x.shape
```

```
torch.Size([4, 8, 2])
```

```
# We want x[b,t] = mean_{i<=t} x[b,i]
xbow = torch.zeros((B,T,C))
for b in range(B):
    for t in range(T):
        xprev = x[b,:t+1] # (t,C)
        xbow[b,t] = torch.mean(xprev, 0)
```

```
# version 2: using matrix multiply for a weighted aggregation
wei = torch.tril(torch.ones(T, T))
wei = wei / wei.sum(1, keepdim=True)
xbow2 = wei @ x # (B, T, T) @ (B, T, C) ----> (B, T, C)
torch.allclose(xbow, xbow2)
```

```
False
```

```
# version 3: use Softmax
tril = torch.tril(torch.ones(T, T))
wei = torch.zeros((T,T))
wei = wei.masked_fill(tril == 0, float('-inf'))
wei = F.softmax(wei, dim=-1)
xbow3 = wei @ x
torch.allclose(xbow, xbow3)
```

```
False
```

```
# version 4: self-attention!
torch.manual_seed(1337)
B,T,C = 4,8,32 # batch, time, channels
x = torch.randn(B,T,C)

# let's see a single Head perform self-attention
head_size = 16
key = nn.Linear(C, head_size, bias=False)
query = nn.Linear(C, head_size, bias=False)
value = nn.Linear(C, head_size, bias=False)
k = key(x)   # (B, T, 16)
q = query(x) # (B, T, 16)
wei =  q @ k.transpose(-2, -1) # (B, T, 16) @ (B, 16, T) ---> (B, T, T)

tril = torch.tril(torch.ones(T, T))
#wei = torch.zeros((T,T))
wei = wei.masked_fill(tril == 0, float('-inf'))
wei = F.softmax(wei, dim=-1)

v = value(x)
out = wei @ v
#out = wei @ x

out.shape
```

```
torch.Size([4, 8, 16])
```

```
wei[0]
```

```
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.1574, 0.8426, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.2088, 0.1646, 0.6266, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.5792, 0.1187, 0.1889, 0.1131, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.0294, 0.1052, 0.0469, 0.0276, 0.7909, 0.0000, 0.0000, 0.0000],
        [0.0176, 0.2689, 0.0215, 0.0089, 0.6812, 0.0019, 0.0000, 0.0000],
        [0.1691, 0.4066, 0.0438, 0.0416, 0.1048, 0.2012, 0.0329, 0.0000],
        [0.0210, 0.0843, 0.0555, 0.2297, 0.0573, 0.0709, 0.2423, 0.2391]],
       grad_fn=<SelectBackward0>)
```

Notes:

- Attention is a **communication mechanism**. Can be seen as nodes in a directed graph looking at each other and aggregating information with a weighted sum from all nodes that point to them, with data-dependent weights.
- There is no notion of space. Attention simply acts over a set of vectors. This is why we need to positionally encode tokens.
- Each example across batch dimension is of course processed completely independently and never "talk" to each other
- In an "encoder" attention block just delete the single line that does masking with `tril`, allowing all tokens to communicate. This block here is called a "decoder" attention block because it has triangular masking, and is usually used in autoregressive settings, like language modeling.
- "self-attention" just means that the keys and values are produced from the same source as queries. In "cross-attention", the queries still get produced from x, but the keys and values come from some other, external source (e.g. an encoder module)
- "Scaled" attention additional divides `wei` by 1/sqrt(head_size). This makes it so when input Q,K are unit variance, wei will be unit variance too and Softmax will stay diffuse and not saturate too much. Illustration below

```
k = torch.randn(B,T,head_size)
q = torch.randn(B,T,head_size)
wei = q @ k.transpose(-2, -1) * head_size**-0.5
```

```
k.var()
```

```
tensor(1.0449)
```

```
q.var()
```

```
tensor(1.0700)
```

```
wei.var()
```

```
tensor(1.0918)
```

```
torch.softmax(torch.tensor([0.1, -0.2, 0.3, -0.2, 0.5]), dim=-1)
```

```
tensor([0.1925, 0.1426, 0.2351, 0.1426, 0.2872])
```

```
torch.softmax(torch.tensor([0.1, -0.2, 0.3, -0.2, 0.5])*8, dim=-1) # gets too peaky, converges to one-hot
```

```
tensor([0.0326, 0.0030, 0.1615, 0.0030, 0.8000])
```

```
class LayerNorm1d: # (used to be BatchNorm1d)

  def __init__(self, dim, eps=1e-5, momentum=0.1):
    self.eps = eps
    self.gamma = torch.ones(dim)
    self.beta = torch.zeros(dim)

  def __call__(self, x):
    # calculate the forward pass
    xmean = x.mean(1, keepdim=True) # batch mean
    xvar = x.var(1, keepdim=True) # batch variance
    xhat = (x - xmean) / torch.sqrt(xvar + self.eps) # normalize to unit variance
    self.out = self.gamma * xhat + self.beta
    return self.out

  def parameters(self):
    return [self.gamma, self.beta]

torch.manual_seed(1337)
module = LayerNorm1d(100)
x = torch.randn(32, 100) # batch size 32 of 100-dimensional vectors
x = module(x)
x.shape
```

```
torch.Size([32, 100])
```

```
x[:,0].mean(), x[:,0].std() # mean,std of one feature across all batch inputs
```

```
(tensor(0.1469), tensor(0.8803))
```

```
x[0,:].mean(), x[0,:].std() # mean,std of a single input from the batch, of its features
```

```
(tensor(-9.5367e-09), tensor(1.0000))
```

```
# French to English translation example:

# <--------- ENCODE ------------------><--------------- DECODE ----------------->
# les réseaux de neurones sont géniaux! <START> neural networks are awesome!<END>
```

3. Encoder language model

To run this section of the notebook, you need a GPU (you can run it on CPU but it will take a much longer). You can run it in Google Colab, and from the menu select **Runtime/Change runtime type** and select **T4 GPU** for **Hardware accelerator**. Then click **Save**.

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

# hyperparameters
batch_size = 16 # how many independent sequences will we process in parallel?
block_size = 32 # what is the maximum context length for predictions?
max_iters = 5000
eval_interval = 100
learning_rate = 1e-1
device = 'cuda' if torch.cuda.is_available() else 'cpu'
eval_iters = 200
n_embd = 64
n_head = 4
n_layer = 4
dropout = 0.0
# ------------

torch.manual_seed(1337)

# wget https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt
# You will have to change the path to the 'input.txt' file (e.g., /content/input.txt)
with open('input.txt', 'r', encoding='utf-8') as f:
    text = f.read()

# here are all the unique characters that occur in this text
chars = sorted(list(set(text)))
vocab_size = len(chars)
# create a mapping from characters to integers
stoi = { ch:i for i,ch in enumerate(chars) }
itos = { i:ch for i,ch in enumerate(chars) }
encode = lambda s: [stoi[c] for c in s] # encoder: take a string, output a list of integers
decode = lambda l: ''.join([itos[i] for i in l]) # decoder: take a list of integers, output a string

# Train and test splits
data = torch.tensor(encode(text), dtype=torch.long)
n = int(0.9*len(data)) # first 90% will be train, rest val
train_data = data[:n]
val_data = data[n:]

# data loading
def get_batch(split):
    # generate a small batch of data of inputs x and targets y
    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

@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

class Head(nn.Module):
    """ one head of self-attention """

    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
        k = self.key(x)   # (B,T,C)
        q = self.query(x) # (B,T,C)
        # compute attention scores ("affinities")
        wei = q @ k.transpose(-2,-1) * C**-0.5 # (B, T, C) @ (B, C, T) -> (B, T, T)
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf')) # (B, T, T)
        wei = F.softmax(wei, dim=-1) # (B, T, T)
        wei = self.dropout(wei)
        # perform the weighted aggregation of the values
        v = self.value(x) # (B,T,C)
        out = wei @ v # (B, T, T) @ (B, T, C) -> (B, T, C)
        return out

class MultiHeadAttention(nn.Module):
    """ multiple heads of self-attention in parallel """

    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(n_embd, n_embd)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        out = torch.cat([h(x) for h in self.heads], dim=-1)
        out = self.dropout(self.proj(out))
        return out

class FeedFoward(nn.Module):
    """ a simple linear layer followed by a non-linearity """

    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 Block(nn.Module):
    """ Transformer block: communication followed by computation """

    def __init__(self, n_embd, n_head):
        # n_embd: embedding dimension, n_head: the number of heads we'd like
        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))
        x = x + self.ffwd(self.ln2(x))
        return x

# super simple bigram model
class BigramLanguageModel(nn.Module):

    def __init__(self):
        super().__init__()
        # each token directly reads off the logits for the next token from a lookup table
        self.token_embedding_table = nn.Embedding(vocab_size, n_embd)
        self.position_embedding_table = nn.Embedding(block_size, n_embd)
        self.blocks = nn.Sequential(*[Block(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)

    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:
            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):
        # idx is (B, T) array of indices in the current context
        for _ in range(max_new_tokens):
            # crop idx to the last block_size tokens
            idx_cond = idx[:, -block_size:]
            # get the predictions
            logits, loss = self(idx_cond)
            # focus only on the last time step
            logits = logits[:, -1, :] # becomes (B, C)
            # apply softmax to get probabilities
            probs = F.softmax(logits, dim=-1) # (B, C)
            # sample from the distribution
            idx_next = torch.multinomial(probs, num_samples=1) # (B, 1)
            # append sampled index to the running sequence
            idx = torch.cat((idx, idx_next), dim=1) # (B, T+1)
        return idx

model = BigramLanguageModel()
m = model.to(device)
# print the number of parameters in the model
print(sum(p.numel() for p in m.parameters())/1e6, 'M parameters')

# create a PyTorch optimizer
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)

for iter in range(max_iters):

    # 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()

# generate from the model
```

```
context = torch.zeros((1, 1), dtype=torch.long, device=device)
print(decode(m.generate(context, max_new_tokens=2000)[0].tolist()))
```

```
0.209729 M parameters
step 0: train loss 4.4116, val loss 4.4022
step 100: train loss 3.3430, val loss 3.3801
step 200: train loss 3.2708, val loss 3.2961
step 300: train loss 3.2503, val loss 3.2806
step 400: train loss 3.2896, val loss 3.3235
step 500: train loss 3.2229, val loss 3.2383
step 600: train loss 3.2251, val loss 3.2533
step 700: train loss 3.2566, val loss 3.2902
step 800: train loss 3.2901, val loss 3.3268
step 900: train loss 3.3005, val loss 3.3322
step 1000: train loss 3.2894, val loss 3.3212
step 1100: train loss 3.2739, val loss 3.3163
step 1200: train loss 3.2855, val loss 3.3188
step 1300: train loss 3.3203, val loss 3.3482
step 1400: train loss 3.2835, val loss 3.3053
step 1500: train loss 3.2767, val loss 3.3072
step 1600: train loss 3.2811, val loss 3.3281
step 1700: train loss 3.2823, val loss 3.3210
step 1800: train loss 3.2458, val loss 3.2726
step 1900: train loss 3.2538, val loss 3.2715
step 2000: train loss 3.2682, val loss 3.3003
step 2100: train loss 3.2839, val loss 3.3094
step 2200: train loss 3.3166, val loss 3.3457
step 2300: train loss 3.3018, val loss 3.3295
step 2400: train loss 3.2746, val loss 3.3125
step 2500: train loss 3.2627, val loss 3.2889
step 2600: train loss 3.2749, val loss 3.3153
step 2700: train loss 3.2452, val loss 3.2499
step 2800: train loss 3.2449, val loss 3.2880
step 2900: train loss 3.2670, val loss 3.2940
step 3000: train loss 3.2452, val loss 3.2696
step 3100: train loss 3.2539, val loss 3.2840
step 3200: train loss 3.2742, val loss 3.2923
step 3300: train loss 3.2692, val loss 3.2930
step 3400: train loss 3.2661, val loss 3.3079
step 3500: train loss 3.2647, val loss 3.2929
step 3600: train loss 3.2405, val loss 3.2707
step 3700: train loss 3.2024, val loss 3.2295
step 3800: train loss 3.2169, val loss 3.2406
```

Questions (25 points each)

1. How many attention heads are used in this encoder? 4
2. How many layers are in each feed-forward network? List these layers. 4 Each feed-forward network consists of two linear layers followed by ReLU activation and a dropout layer. 1 Linear layer: input size = n_embd, output size = 4 * n_embd 2 ReLU activation 3 Linear layer: input size = 4 * n_embd, output size = n_embd 4 Dropout layer
3. How many layers are in the transformer block? List these layers. The transformer block consists of two main components: Multi-head self-attention mechanism Feed-forward network Each block includes: Multi-head attention mechanism, Layer normalization, Feed-forward network, Another layer normalization.

Extra credit (10 points)

How does the **learning rate** affect the model (in terms of learning time, and accuracy)? Pick a different value for the learning rate and run the code to show this. A higher learning rate may lead to faster convergence initially, but it might overshoot the optimal solution and cause divergence. Conversely, a lower learning rate may slow down convergence but might yield better accuracy as it allows the model to explore the solution space more thoroughly. I changed the learning time from 1e -3 to 1e -4. the 1e -4 loss average was well above 2.0 the average i would estimate at around 2.4. When i ran 1e - 2 the average train loss i would estimate at around 1.8 and var loss at around 2. From an eyeball test the faster learning rate (1e - 2) was much more readable and comprehensible verues the 1e -4 which looked like a combination of letters.

In this case it seams slowing down convergence leads to a less accurate result