Building a GPT

Companion notebook to the Zero To Hero video on GPT.

In [1]:
```
# NOTES UP TO SELF ATTENTION

# We always start with a dataset to train on. Let's download the tiny shakespeare dataset
!wget https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt
```

```
--2023-05-21 00:23:31--  https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.
txt
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 185.199.109.133, 185.199.11
0.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1115394 (1.1M) [text/plain]
Saving to: ‘input.txt’

input.txt           100%[===================>]   1.06M  --.-KB/s    in 0.06s

2023-05-21 00:23:31 (18.4 MB/s) - ‘input.txt’ saved [1115394/1115394]
```

In [2]:
```
# read it in to inspect it
with open('input.txt', 'r', encoding='utf-8') as f:
    text = f.read()
```

In [3]:
```
print("length of dataset in characters: ", len(text))
```

```
length of dataset in characters:  1115394
```

In [4]:
```
# let's look at the first 1000 characters
print(text[:1000])
```

```
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.

First Citizen:
First, you know Caius Marcius is chief enemy to the people.

All:
We know't, we know't.

First Citizen:
Let us kill him, and we'll have corn at our own price.
Is't a verdict?

All:
No more talking on't; let it be done: away, away!

Second Citizen:
One word, good citizens.

First Citizen:
We are accounted poor citizens, the patricians good.
What authority surfeits on would relieve us: if they
would yield us but the superfluity, while it were
wholesome, we might guess they relieved us humanely;
but they think we are too dear: the leanness that
afflicts us, the object of our misery, is as an
inventory to particularise their abundance; our
sufferance is a gain to them Let us revenge this with
our pikes, ere we become rakes: for the gods know I
speak this in hunger for bread, not in thirst for revenge.
```

In [5]:
```
# 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)
```

```
 !$&',-.3:;?ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
65
```

In [6]:

```
# create a mapping from characters to integers
stoi = { ch:i for i,ch in enumerate(chars) }  # create a lookup table from char to integer
itos = { i:ch for i,ch in enumerate(chars) }
# to encode + decode, translate all characters individually
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")))
```

```
[46, 47, 47, 1, 58, 46, 43, 56, 43]
hii there
```

In [7]:

```
# 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
```

```
torch.Size([1115394]) torch.int64
tensor([18, 47, 56, 57, 58,  1, 15, 47, 58, 47, 64, 43, 52, 10,  0, 14, 43, 44,
        53, 56, 43,  1, 61, 43,  1, 54, 56, 53, 41, 43, 43, 42,  1, 39, 52, 63,
         1, 44, 59, 56, 58, 46, 43, 56,  6,  1, 46, 43, 39, 56,  1, 51, 43,  1,
        57, 54, 43, 39, 49,  8,  0,  0, 13, 50, 50, 10,  0, 31, 54, 43, 39, 49,
         6,  1, 57, 54, 43, 39, 49,  8,  0,  0, 18, 47, 56, 57, 58,  1, 15, 47,
        58, 47, 64, 43, 52, 10,  0, 37, 53, 59,  1, 39, 56, 43,  1, 39, 50, 50,
         1, 56, 43, 57, 53, 50, 60, 43, 42,  1, 56, 39, 58, 46, 43, 56,  1, 58,
        53,  1, 42, 47, 43,  1, 58, 46, 39, 52,  1, 58, 53,  1, 44, 39, 51, 47,
        57, 46, 12,  0,  0, 13, 50, 50, 10,  0, 30, 43, 57, 53, 50, 60, 43, 42,
         8,  1, 56, 43, 57, 53, 50, 60, 43, 42,  8,  0,  0, 18, 47, 56, 57, 58,
         1, 15, 47, 58, 47, 64, 43, 52, 10,  0, 18, 47, 56, 57, 58,  6,  1, 63,
        53, 59,  1, 49, 52, 53, 61,  1, 15, 39, 47, 59, 57,  1, 25, 39, 56, 41,
        47, 59, 57,  1, 47, 57,  1, 41, 46, 47, 43, 44,  1, 43, 52, 43, 51, 63,
         1, 58, 53,  1, 58, 46, 43,  1, 54, 43, 53, 54, 50, 43,  8,  0,  0, 13,
        50, 50, 10,  0, 35, 43,  1, 49, 52, 53, 61,  5, 58,  6,  1, 61, 43,  1,
        49, 52, 53, 61,  5, 58,  8,  0,  0, 18, 47, 56, 57, 58,  1, 15, 47, 58,
        47, 64, 43, 52, 10,  0, 24, 43, 58,  1, 59, 57,  1, 49, 47, 50, 50,  1,
        46, 47, 51,  6,  1, 39, 52, 42,  1, 61, 43,  5, 50, 50,  1, 46, 39, 60,
        43,  1, 41, 53, 56, 52,  1, 39, 58,  1, 53, 59, 56,  1, 53, 61, 52,  1,
        54, 56, 47, 41, 43,  8,  0, 21, 57,  5, 58,  1, 39,  1, 60, 43, 56, 42,
        47, 41, 58, 12,  0,  0, 13, 50, 50, 10,  0, 26, 53,  1, 51, 53, 56, 43,
         1, 58, 39, 50, 49, 47, 52, 45,  1, 53, 52,  5, 58, 11,  1, 50, 43, 58,
         1, 47, 58,  1, 40, 43,  1, 42, 53, 52, 43, 10,  1, 39, 61, 39, 63,  6,
         1, 39, 61, 39, 63,  2,  0,  0, 31, 43, 41, 53, 52, 42,  1, 15, 47, 58,
        47, 64, 43, 52, 10,  0, 27, 52, 43,  1, 61, 53, 56, 42,  6,  1, 45, 53,
        53, 42,  1, 41, 47, 58, 47, 64, 43, 52, 57,  8,  0,  0, 18, 47, 56, 57,
        58,  1, 15, 47, 58, 47, 64, 43, 52, 10,  0, 35, 43,  1, 39, 56, 43,  1,
        39, 41, 41, 53, 59, 52, 58, 43, 42,  1, 54, 53, 53, 56,  1, 41, 47, 58,
        47, 64, 43, 52, 57,  6,  1, 58, 46, 43,  1, 54, 39, 58, 56, 47, 41, 47,
        39, 52, 57,  1, 45, 53, 53, 42,  8,  0, 35, 46, 39, 58,  1, 39, 59, 58,
        46, 53, 56, 47, 58, 63,  1, 57, 59, 56, 44, 43, 47, 58, 57,  1, 53, 52,
         1, 61, 53, 59, 50, 42,  1, 56, 43, 50, 47, 43, 60, 43,  1, 59, 57, 10,
         1, 47, 44,  1, 58, 46, 43, 63,  0, 61, 53, 59, 50, 42,  1, 63, 47, 43,
        50, 42,  1, 59, 57,  1, 40, 59, 58,  1, 58, 46, 43,  1, 57, 59, 54, 43,
        56, 44, 50, 59, 47, 58, 63,  6,  1, 61, 46, 47, 50, 43,  1, 47, 58,  1,
        61, 43, 56, 43,  0, 61, 46, 53, 50, 43, 57, 53, 51, 43,  6,  1, 61, 43,
         1, 51, 47, 45, 46, 58,  1, 45, 59, 43, 57, 57,  1, 58, 46, 43, 63,  1,
        56, 43, 50, 47, 43, 60, 43, 42,  1, 59, 57,  1, 46, 59, 51, 39, 52, 43,
        50, 63, 11,  0, 40, 59, 58,  1, 58, 46, 43, 63,  1, 58, 46, 47, 52, 49,
         1, 61, 43,  1, 39, 56, 43,  1, 58, 53, 53,  1, 42, 43, 39, 56, 10,  1,
        58, 46, 43,  1, 50, 43, 39, 52, 52, 43, 57, 57,  1, 58, 46, 39, 58,  0,
        39, 44, 44, 50, 47, 41, 58, 57,  1, 59, 57,  6,  1, 58, 46, 43,  1, 53,
        40, 48, 43, 41, 58,  1, 53, 44,  1, 53, 59, 56,  1, 51, 47, 57, 43, 56,
        63,  6,  1, 47, 57,  1, 39, 57,  1, 39, 52,  0, 47, 52, 60, 43, 52, 58,
        53, 56, 63,  1, 58, 53,  1, 54, 39, 56, 58, 47, 41, 59, 50, 39, 56, 47,
        57, 43,  1, 58, 46, 43, 47, 56,  1, 39, 40, 59, 52, 42, 39, 52, 41, 43,
        11,  1, 53, 59, 56,  0, 57, 59, 44, 44, 43, 56, 39, 52, 41, 43,  1, 47,
        57,  1, 39,  1, 45, 39, 47, 52,  1, 58, 53,  1, 58, 46, 43, 51,  1, 24,
        43, 58,  1, 59, 57,  1, 56, 43, 60, 43, 52, 45, 43,  1, 58, 46, 47, 57,
         1, 61, 47, 58, 46,  0, 53, 59, 56,  1, 54, 47, 49, 43, 57,  6,  1, 43,
        56, 43,  1, 61, 43,  1, 40, 43, 41, 53, 51, 43,  1, 56, 39, 49, 43, 57,
        10,  1, 44, 53, 56,  1, 58, 46, 43,  1, 45, 53, 42, 57,  1, 49, 52, 53,
        61,  1, 21,  0, 57, 54, 43, 39, 49,  1, 58, 46, 47, 57,  1, 47, 52,  1,
        46, 59, 52, 45, 43, 56,  1, 44, 53, 56,  1, 40, 56, 43, 39, 42,  6,  1,
        52, 53, 58,  1, 47, 52,  1, 58, 46, 47, 56, 57, 58,  1, 44, 53, 56,  1,
        56, 43, 60, 43, 52, 45, 43,  8,  0,  0])
```

In [8]:

```
# 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:]
```

In [9]:
```
block_size = 8
train_data[:block_size+1]
```

Out[9]:
```
tensor([18, 47, 56, 57, 58,  1, 15, 47, 58])
```

In [10]:
```
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([18]) the target: 47
when input is tensor([18, 47]) the target: 56
when input is tensor([18, 47, 56]) the target: 57
when input is tensor([18, 47, 56, 57]) the target: 58
when input is tensor([18, 47, 56, 57, 58]) the target: 1
when input is tensor([18, 47, 56, 57, 58,  1]) the target: 15
when input is tensor([18, 47, 56, 57, 58,  1, 15]) the target: 47
when input is tensor([18, 47, 56, 57, 58,  1, 15, 47]) the target: 58
```

In [11]:
```
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([[24, 43, 58,  5, 57,  1, 46, 43],
        [44, 53, 56,  1, 58, 46, 39, 58],
        [52, 58,  1, 58, 46, 39, 58,  1],
        [25, 17, 27, 10,  0, 21,  1, 54]])
targets:
torch.Size([4, 8])
tensor([[43, 58,  5, 57,  1, 46, 43, 39],
        [53, 56,  1, 58, 46, 39, 58,  1],
        [58,  1, 58, 46, 39, 58,  1, 46],
        [17, 27, 10,  0, 21,  1, 54, 39]])
----
when input is [24] the target: 43
when input is [24, 43] the target: 58
when input is [24, 43, 58] the target: 5
when input is [24, 43, 58, 5] the target: 57
when input is [24, 43, 58, 5, 57] the target: 1
when input is [24, 43, 58, 5, 57, 1] the target: 46
when input is [24, 43, 58, 5, 57, 1, 46] the target: 43
when input is [24, 43, 58, 5, 57, 1, 46, 43] the target: 39
when input is [44] the target: 53
when input is [44, 53] the target: 56
when input is [44, 53, 56] the target: 1
when input is [44, 53, 56, 1] the target: 58
when input is [44, 53, 56, 1, 58] the target: 46
when input is [44, 53, 56, 1, 58, 46] the target: 39
when input is [44, 53, 56, 1, 58, 46, 39] the target: 58
when input is [44, 53, 56, 1, 58, 46, 39, 58] the target: 1
when input is [52] the target: 58
when input is [52, 58] the target: 1
when input is [52, 58, 1] the target: 58
when input is [52, 58, 1, 58] the target: 46
when input is [52, 58, 1, 58, 46] the target: 39
when input is [52, 58, 1, 58, 46, 39] the target: 58
when input is [52, 58, 1, 58, 46, 39, 58] the target: 1
when input is [52, 58, 1, 58, 46, 39, 58, 1] the target: 46
when input is [25] the target: 17
when input is [25, 17] the target: 27
when input is [25, 17, 27] the target: 10
when input is [25, 17, 27, 10] the target: 0
when input is [25, 17, 27, 10, 0] the target: 21
when input is [25, 17, 27, 10, 0, 21] the target: 1
when input is [25, 17, 27, 10, 0, 21, 1] the target: 54
when input is [25, 17, 27, 10, 0, 21, 1, 54] the target: 39
```

In [12]:

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

```
tensor([[24, 43, 58,  5, 57,  1, 46, 43],
        [44, 53, 56,  1, 58, 46, 39, 58],
        [52, 58,  1, 58, 46, 39, 58,  1],
        [25, 17, 27, 10,  0, 21,  1, 54]])
```

In [13]:

```python
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, 65])
tensor(4.8786, grad_fn=<NllLossBackward0>)

Sr?qP-QWktXoL&jLDJgOLVz'RIoDqHdhsV&vLLxatjscMpwLERSPyao.qfzs$Ys$zF-w,;eEkzxjgCKFChs!iWW.ObzDnxA Ms$3
```

In [14]:
```
# create a PyTorch optimizer
optimizer = torch.optim.AdamW(m.parameters(), lr=1e-3)
```

In [15]:
```
batch_size = 32
for steps in range(100): # 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())
```

```
4.587916374206543
```

In [16]:
```
print(decode(m.generate(idx = torch.zeros((1, 1), dtype=torch.long), max_new_tokens=500)[0].tolist()))
```

```
xiKi-RJ:CgqVuUa!U?qMH.uk!sCuMXvv!CJFfx;LgRyJknOEti.?I&-gPlLyulId?XlaInQ'q,lT$
3Q&sGlvHQ?mqSq-eON
x?SP fUAfCAuCX:bOlgiRQWN:Mphaw
tRLKuYXEaAXxrcq-gCUzeh3w!AcyaylgYWjmJM?Uzw:inaY,:C&OECW:vmGGJAn3onAuMgia!ms$Vb q-gCOcPcUhOnxJGUGSPJWT:.?ujmJFoi
NL&A'DxY,prZ?qdT;hoo'dHooXXlxf'WkHK&u3Q?rqUi.kz;?Yx?C&u3Qbfzxlyh'Vl:zyxjKXgC?
lv'QKFiBeviNxO'm!Upm$srm&TqViqiBD3HBP!juEOpmZJyF$Fwfy!PlvWPFC
&WDdP!Ko,px
x
tREOE;AJ.BeXkylOVD3KHp$e?nD,.SFbWWI'ubcL!q-tU;aXmJ&uGXHxJXI&Z!gHRpajj;l.
pTErIBjx;JKIgoCnLGXrJSP!AU-AcbczR?
```

The mathematical trick in self-attention

In [17]:
```
# 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]])
```

In [18]:

```
# 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
```

Out[18]:

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

In [20]:

```
# We want x[b,t] = mean_{i<=t} x[b,i]
xbow = torch.zeros((B,T,C))   # bow = bag of words
for b in range(B):            # over batch dimensions
    for t in range(T):        # over time
        xprev = x[b,:t+1]     # everything up to and including t token, xprev becomes shape (t,C)
        xbow[b,t] = torch.mean(xprev, 0)

# inefficient, can be made more efficient using matrix multiplication
```

In [21]:

```
# version 2: using matrix multiply for a weighted aggregation
# tril returns lower triangular portion of matrix, zeroes out unused elements
wei = torch.tril(torch.ones(T, T))    # wei short for weights
wei = wei / wei.sum(1, keepdim=True)  # all elements in a row of wei sum to 1
# when multiplied, xbow2 takes average of all wei's previous rows
# @ will see wei is (T,T) and create B (batch) dimension for batch matrix multiplication
# applies matrix multiplication in all batch elements in parallel
xbow2 = wei @ x # (B, T, T) @ (B, T, C) ----> (B, T, C)
torch.allclose(xbow, xbow2)
```

Out[21]:

```
False
```

In [24]:

```
# version 3: use Softmax
tril = torch.tril(torch.ones(T, T))
wei = torch.zeros((T,T))    # how much weight does each previous token get and avg. up
wei = wei.masked_fill(tril == 0, float('-inf'))   # future tokens cannot communicate with the past
# softmax exponentiates each element and divides by sum
wei = F.softmax(wei, dim=-1)
xbow3 = wei @ x
# aggregation (structure) via matrix multiplication, inifinities between 0's are
# data dependent, tokens start looking at each other and derive levels of interest (affinity)
# when we normalize and sum, we get weighted aggregation of past elements by using
# matrix multiplication of lower triangular fashion, elements say how much to fuse
# to this position. This is the basis of self-attention.
torch.allclose(xbow, xbow3)
```

Out[24]:

```
False
```

In [ ]:

```
# 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
# each token has a key (info so far) and query (info wanted) and dot product of these tells weights
# eg. 8th token knows what content it has and its position, creates a query
# all tokens emit keys, 1 channel/token could satisfy query, so dot product of its key and query will yield higl
# through softmax, a lot of its info will be given (aka its weight) into that tokens position (eg. 4th token)
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)
# @ is dot product or pooling, transpose the last 2 dimensions of key
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'))     # zeros out future tokens/lower triangle
wei = F.softmax(wei, dim=-1)

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

out.shape
```

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

In [ ]:
```
wei[0]
```

Out[ ]:
```
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

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

In [ ]:
```
k.var()
```

Out[ ]:
```
tensor(1.0449)
```

In [ ]:
```
q.var()
```

Out[ ]:
```
tensor(1.0700)
```

In [ ]:
```
wei.var()
```

Out[ ]:
```
tensor(1.0918)
```

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

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

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

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

In [ ]:
```
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
    # normalize rows to mimic layernorm function
    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
```

Out[ ]: torch.Size([32, 100])

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

Out[ ]: (tensor(0.1469), tensor(0.8803))

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

Out[ ]: (tensor(-9.5367e-09), tensor(1.0000))

In [ ]:
```
# French to English translation example:

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

Full finished code, for reference

You may want to refer directly to the git repo instead though.

In [ ]:
```
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-3
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
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']:
```

```python
        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)
        # tril is not a parameter of the module, so in pytorch naming conventions it is a buffer
        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")
        # scale it by 1/sqrt(head size) to have variance of weights be within 1, which is fed into softmax
        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):
        # concatenate all head outputs over channel dimension
        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), # in paper, inner layer had 4x input dim
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd), # projection layer going back to residual pathway
            # can add dropout right before connecting back to residual pathway
            # after every forward and back pass, dropoff shuts off (zeros out) random subset of neurons
            # trains without them, bc mask of whats being zerod out changes every pass, it ends up
            # training an ensemble of sub-networks, at test time everything is enabled, and sub-networks are me
            # hyperparameter dropout controls % of zero'ing out neurons (eg. 20%)
            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)
```

```python
    def forward(self, x):
        # in original transformer paper, add & norm (layernorm layer) occured after transformation
        # now its more normal to do it before self-attention and feed-forward layers
        x = x + self.sa(self.ln1(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)
        # to scale up model, n_layer specifies how many layers of blocks
        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
        # from token embeddings to logits we need a linear layer
        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()
```

```python
# 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 2.6568, val loss 2.6670
step 200: train loss 2.5090, val loss 2.5058
step 300: train loss 2.4198, val loss 2.4340
step 400: train loss 2.3503, val loss 2.3567
step 500: train loss 2.2970, val loss 2.3136
step 600: train loss 2.2410, val loss 2.2506
step 700: train loss 2.2062, val loss 2.2198
step 800: train loss 2.1638, val loss 2.1871
step 900: train loss 2.1232, val loss 2.1494
step 1000: train loss 2.1020, val loss 2.1293
step 1100: train loss 2.0704, val loss 2.1196
step 1200: train loss 2.0382, val loss 2.0798
step 1300: train loss 2.0249, val loss 2.0640
step 1400: train loss 1.9922, val loss 2.0354
step 1500: train loss 1.9707, val loss 2.0308
step 1600: train loss 1.9614, val loss 2.0474
step 1700: train loss 1.9393, val loss 2.0130
step 1800: train loss 1.9070, val loss 1.9943
step 1900: train loss 1.9057, val loss 1.9871
step 2000: train loss 1.8834, val loss 1.9954
step 2100: train loss 1.8719, val loss 1.9758
step 2200: train loss 1.8582, val loss 1.9623
step 2300: train loss 1.8546, val loss 1.9517
step 2400: train loss 1.8410, val loss 1.9476
step 2500: train loss 1.8167, val loss 1.9455
step 2600: train loss 1.8263, val loss 1.9401
step 2700: train loss 1.8108, val loss 1.9340
step 2800: train loss 1.8040, val loss 1.9247
step 2900: train loss 1.8044, val loss 1.9304
step 3000: train loss 1.7963, val loss 1.9242
step 3100: train loss 1.7687, val loss 1.9147
step 3200: train loss 1.7547, val loss 1.9102
step 3300: train loss 1.7557, val loss 1.9037
step 3400: train loss 1.7547, val loss 1.8946
step 3500: train loss 1.7385, val loss 1.8968
step 3600: train loss 1.7260, val loss 1.8914
step 3700: train loss 1.7257, val loss 1.8808
step 3800: train loss 1.7204, val loss 1.8919
step 3900: train loss 1.7215, val loss 1.8788
step 4000: train loss 1.7146, val loss 1.8639
step 4100: train loss 1.7095, val loss 1.8724
step 4200: train loss 1.7079, val loss 1.8707
step 4300: train loss 1.7035, val loss 1.8502
step 4400: train loss 1.7043, val loss 1.8693
step 4500: train loss 1.6914, val loss 1.8522
step 4600: train loss 1.6853, val loss 1.8357
step 4700: train loss 1.6862, val loss 1.8483
step 4800: train loss 1.6671, val loss 1.8434
step 4900: train loss 1.6736, val loss 1.8415
step 4999: train loss 1.6635, val loss 1.8226

FlY BOLINGLO:
Them thrumply towiter arts the
muscue rike begatt the sea it
What satell in rowers that some than othis Marrity.

LUCENTVO:
But userman these that, where can is not diesty rege;
What and see to not. But's eyes. What?

JOHN MARGARET:
Than up I wark, what out, I ever of and love,
one these do sponce, vois I me;
But my pray sape to ries all to the not erralied in may.

BENVOLIO:
To spits as stold's bewear I would and say mesby all
on sworn make he anough
As cousins the solle, whose be my conforeful may lie them yet
nobe allimely untraled to be thre I say be,
Notham a brotes theme an make come,
And that his reach to the duke ento
the grmeants bell! and now there king-liff-or grief?

GLOUCESTER:
All the bettle dreene, for To his like thou thron!
```

```
MENENIUS:
Then, if I knom her all.
My lord, but terruly friend
Rish of the ploceiness and wilt tends sure?
Is you knows a fasir wead
That with him my spaut,
I shall not tas where's not, becomity; my coulds sting,
then the wit be dong to tyget our hereefore,
Who strop me, mend here, if agains, bitten, thy lack.
The but these it were is tus. For the her skeep the fasting. joy tweet Bumner:-
How the enclady: It you and how,
I am in him, And ladderle:
Their hand whose wife, it my hithre,
Roman and where sposs gives'd you.

TROMIOLANUS:
But livants you great, I shom mistrot come, for to she to lot
for smy to men ventry mehus. Gazise;
Full't were some the cause, and stouch set,
Or promises, which a kingsasted to your gove them; and sterrer,
And that wae love him.

BRUTUS:
You shape with these sweet.

CORTENGONO:
Lo, where 'twon elmes, 'morth young agres;
Sir, azavoust to striel accurded we missery sets crave.

ANGOLUM:
For is Henry to have gleise the dreason
That I ant shorfold wefth their servy in enscy.

ISABELLA:
O, I better you eyse such formfetrews.

BUCKINGHARENT:
Qead my lightle this righanneds flase them
Wam which an take was our some pleasurs,
Lovisoname to me, then fult me?--have it?

HENRY BOLINGBROY:
That wha
```

In [ ]: