Basic_transformer(1)

September 30, 2023

[1]:
```python
import torch
import torch.nn as nn
from torch.nn import functional as F
```

[2]:
```python
max_iters = 5000
eval_interval = 500
learning_rate = 3e-4
device = 'cuda' if torch.cuda.is_available() else 'cpu'
eval_iters = 200
block_size = 256
batch_size = 64
n_head = 6
n_layer = 6
n_embd = 384
dropout = 0.2
```

This follows a tutorial for "tiny shakespear" by Andrej Karpathy and is not my own work

[3]:
```python
with open('input.txt', 'r', encoding='utf-8') as f:
    text = f.read()
```

[4]:
```python
print(len(text))
```

```
1115394
```

[23]:
```python
print(text[:100])
```

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

All:
Speak, speak.

First Citizen:
You
```

[6]:
```python
chars = sorted(set(text))
vocab_size = len(chars)
print(chars, vocab_size)
```

```
['\n', ' ', '!', '$', '&', "'", ',', '-', '.', '3', ':', ';', '?', 'A', 'B',
'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R',
'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',
'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x',
'y', 'z'] 65
```

Making an "encoder" and decoder to turn chars to ints and then ints to chars

[7]:
```
s_to_i = { ch:i for i,ch in enumerate(chars)}
i_to_s = { i:ch for i, ch in enumerate(chars)}

encode = lambda s: [s_to_i[c] for c in s]
decode = lambda l: ''.join([i_to_s[i] for i in l])

secret = encode("hello my name is tinyshake!")
print(secret)
```

```
[46, 43, 50, 50, 53, 1, 51, 63, 1, 52, 39, 51, 43, 1, 47, 57, 1, 58, 47, 52, 63,
57, 46, 39, 49, 43, 2]
```

[8]:
```
print(decode(secret))
```

```
hello my name is tinyshake!
```

[24]:
```
data = torch.tensor(encode(text), dtype=torch.long)
print(data.shape, data.dtype)
print(data[:100])
```

```
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])
```

[10]:
```
#train and validation split
n = int(0.9*len(data))
train_data = data[:n]
val_data = data[:n]
```

[11]:
```
train_data[:block_size +1]
```

[11]:
```
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])
```

1 Testing!

[12]:
```python
# 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("when we input", context, "we ant to get", target)
```

[13]:
```python
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

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

```
inputs:
torch.Size([64, 256])
```

[14]:
```python
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
        #Single head self attention

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

        wei = q @ k.transpose(-2, -1) * C**-0.5
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf')) # this is a decoder, this means that if you remove this, it becomes an encoder, aka, it can talk to nodes in the future
        wei = F.softmax(wei, dim=-1)
        wei = self.droput(wei)
        v = self.value(x)
        out = wei @ v

        return out

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(num_heads * head_size, 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 FeedForward(nn.Module):
    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):
```

```python
    def __init__(self, n_embd, n_head):
        super().__init__()
        head_size = n_embd // n_head
        self.sa = MultiHeadAttention(n_head, head_size)
        self.fwd = FeedForward(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.fwd(self.ln2(x))
        return x

class BigramLanguageModel(nn.Module):

    def __init__(self):
        super().__init__()
        self.token_embeding_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) for _ in range(n_layer)])
        self.ln_f = nn.LayerNorm(n_embd)
        self.lm_head = nn.Linear(n_embd, vocab_size)

    def forward(self, idx, targets=None):
        B, T = idx.shape
        token_emb = self.token_embeding_table(idx)
        pos_emb = self.position_embedding_table(torch.arange(T, device=device))
        x = token_emb + pos_emb
        x = self.blocks(x)
        logits = self.lm_head(x)

        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):
```

```python
        for _ in range(max_new_tokens):
            idx_cond = idx[:, -block_size:]

            logits, loss = self(idx_cond)

            logits = logits[:, -1, :]

            probs = F.softmax(logits, dim=1)

            idx_next = torch.multinomial(probs, num_samples=1)

            idx = torch.cat((idx, idx_next), dim=1)
        return idx

model = BigramLanguageModel()
m = model.to(device)

@torch.no_grad()
def estimate_loss():
    out = {}
    m.eval()

    for split in ['train', 'val']:
        losses = torch.zeros(eval_iters)
        for k in range(eval_iters):
            X, Y = get_batch(split)
            logits, loss = m(X, Y)
            losses[k] = loss.item()
        out[split] = losses.mean()
    m.train()
    return out
```

1.1 Train the model!

[15]:
```python
optimizer = torch.optim.Adam(m.parameters(), lr=learning_rate)
```

[16]:
```python
for iter in range(max_iters):

    if iter % eval_interval == 0:
        losses = estimate_loss()
        print(f"step {iter}: train loss {losses['train']:.4f}, val loss␣
↪{losses['val']:.4f}")

    xb, yb = get_batch('train')

    logits, loss = m(xb, yb)
    optimizer.zero_grad(set_to_none=True)
```

```
    loss.backward()
    optimizer.step()

print(loss.item())
```

```
step 0: train loss 4.5498, val loss 4.5503
step 500: train loss 2.1003, val loss 2.1019
step 1000: train loss 1.6794, val loss 1.6755
step 1500: train loss 1.5011, val loss 1.4987
step 2000: train loss 1.3951, val loss 1.3951
step 2500: train loss 1.3255, val loss 1.3264
step 3000: train loss 1.2743, val loss 1.2754
step 3500: train loss 1.2285, val loss 1.2286
step 4000: train loss 1.1869, val loss 1.1878
step 4500: train loss 1.1545, val loss 1.1524
1.207191824913025
```

[17]:
```
context = torch.zeros((1, 1), dtype=torch.long, device=device)
print(decode(m.generate(context, max_new_tokens=500)[0].tolist()))
```

```

Half they on the nobles the stumpeth;
Come, let the favour than my hrdship a poor.

LUCIO:
Suspers, go burn!

TRANIS:
Do you rather?

DUKE VINCENTIO:
As you do, and I will desire our gentleman live.

POLIXENES:
Then me right us: he shall be my us the wife,
And give me him for Angelo.

DUKE VINCENTIO:
When he consuled you this armour's Christent's great daughter's.

LUCIO:
Cominiul too!

DUKE VINCENTIO:
If I have proved to a upon two are
Precial that reapt me to bosominable to
us we gracely than
```

[18]:
```
torch.manual_seed(1337)
B, T, C = 4, 8, 2
x = torch.randn(B,T,C)
x.shape
```

[18]: torch.Size([4, 8, 2])

[19]:
```
xbow = torch.zeros((B,T,C))
for b in range(B):
    for t in range(T):
        xprev = x[b, :t+1]
        xbow[b,t] = torch.mean(xprev, 0)
```

[20]:
```
wei = torch.tril(torch.ones(T, T))
wei = wei / wei.sum(1, keepdim=True)
xbow2 = wei @ x
```

2 Self-attention (making our heads)

[21]:
```
torch.manual_seed(1337)

B,T, C = 4, 8, 32
x = torch.randn(B, T, C)

#Single head 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)
q = query(x)
wei = q @ k.transpose(-2, -1)

tril = torch.tril(torch.ones(T, T))
wei = wei.masked_fill(tril == 0, float('-inf')) # this is a decoder, this means that if you remove this, it becomes an encoder, aka, it can talk to nodes in the future
wei = F.softmax(wei, dim=-1)

v = value(x)
out = wei @ v

out.shape
```

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