llmP

August 25, 2024

[28]:
```
import torch
import time
import numpy as np
import torch.nn.functional as F
import torch.nn as nn
import mmap
import random
import pickle
import argparse

device = 'mps' if torch.backends.mps.is_available() else 'cpu'
print(f"Using device: {device}")
block_size = 64
batch_size = 130
max_iters = 1000
eval_interval = 500
learning_rate = 3e-4
eval_iters = 100
dropout = 0.2
n_embd = 384
n_layer = 4
n_head = 4
```

```
Using device: mps
```

[29]:
```
chars = ""
with open('vocab.txt', 'r', encoding='utf-8') as f:
    text=f.read()
    chars = sorted(set(text))

vocab_size = len(chars)
```

[30]:
```
string_to_int = {ch:i for i,ch in enumerate(chars)}
int_to_string={i:ch for i,ch in enumerate(chars)}
encode = lambda s: [string_to_int[c] for c in s]
decode = lambda l: ''.join([int_to_string[i] for i in l])
```

[31]:

```
#memory map for using small snippets of text from a file of any size
def get_random_chunk(split):
    filename = "train_split.txt" if split == 'train' else "val_split.txt"
    with open(filename, 'rb') as f:
        with mmap.mmap(f.fileno(), 0, access=mmap.ACCESS_READ) as mm:
            #determine the file size and a random position to start reading
            file_size = len(mm)
            start_pos= random.randint(0, (file_size) - block_size*batch_size)

            #seek to the random position and read the block of text
            mm.seek(start_pos)
            block = mm.read(block_size*batch_size-1)

            #decode the block to a string, ignoring any invalid bytes sequence
            decoded_block = block.decode('utf-8', errors='ignore').replace('\r','')

            #train and test splits
            data = torch.tensor(encode(decoded_block), dtype=torch.long)

    return data

def get_batch(split):
    data = get_random_chunk(split)
    ix = torch.randint(len(data) - block_size - 1, (batch_size,))  # Adjusted to ensure the sequence length matches
    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
```

[32]:

```
@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
```

[33]:
```
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):
        #input of size (batch, time-step, channels)
        #output of size (batch, time-step, head size)
        B, T, C = x.shape
        k = self.key(x) #(b,t,hs)
        q = self.query(x) #(b,t, hs)
        #compute attention scores("affinities")
        wei = q @ k.transpose(-2, -1) * k.shape[-1]**-0.5 #(b, t, hs) @ (b, hs,␣
↪t) -> (b, t, t)

        if T > block_size:
            tril = torch.tril(torch.ones(T, T, device=x.device))
        else:
            tril = self.tril[:T, :T]

        #wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf')) #(b, t, t)
        wei = wei.masked_fill(tril == 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,hs)
        out = wei @ v #(b, t, t) @ ( b, t, hs) -> (b, t, hs)
        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(head_size * num_heads, n_embd)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
```

```python
        out = torch.cat([h(x) for h in self.heads], dim=-1) #(B, T, Featurs) ->␣
↪(B, T, [h1,h1,h1,h1,h2,h2,h2,h2,h3,h3,h3,h3])
        out = self.dropout(self.proj(out))
        return out

class FeedForward(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 dimensions, 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 = FeedForward(n_embd)
        self.ln1 = nn.LayerNorm(n_embd)
        self.ln2 = nn.LayerNorm(n_embd)
    def forward(self, x):
        y = self.sa(x)
        x = self.ln1(x+y)
        y = self.ffwd(x)
        x = self.ln2(x+y)
        return x


class GPTLanguageModel(nn.Module):
    def __init__(self, vocab_size):
        super().__init__()
        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) #fnal layer norm
        self.lm_head = nn.Linear(n_embd, vocab_size)
```

```python
        self.apply(self.__init__weights)

    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, index, targets=None):
        B, T = index.shape

        #idx and targets are both (B,T) tensor of integers
        tok_emb = self.token_embedding_table(index)
        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, index, max_new_tokens):
        #index is (B,T) array of indices in the current context
        for _ in range(max_new_tokens):
            #get the predictions
            logits, loss = self.forward(index)
            #focus on the last time step
            logits = logits[:,-1,:] #becomes(b,C)
            #apply softmax to get prob
            probs = F.softmax(logits, dim=-1) #(B,C)
            #sample from the distribution
            index_next = torch.multinomial(probs, num_samples=1) #(B,1)
            #append sampled index to the running sequence
            index = torch.cat((index, index_next), dim=1) #(B, T+1)
        return index

model = GPTLanguageModel(vocab_size)
```

```python
print('loading model parameters...')
with open('model-01.pk1', 'rb') as f:
    model = pickle.load(f)
print('loaded successfully')
m = model.to(device)


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

```
loading model parameters…
loaded successfully
```

[34]:
```python
# create a PyTorch optimizer
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)

for iter in range(max_iters):
    if iter % eval_iters == 0:
        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.forward(xb, yb)
    optimizer.zero_grad(set_to_none= True)
    loss.backward()
    optimizer.step()

print(f"Final loss: {loss.item()}")

with open('model-01.pk1', 'wb') as f:
    pickle.dump(model, f)
print('model saved')
```

```
step: 0, train loss1.4892, val loss: 1.4332
step: 100, train loss1.4138, val loss: 1.4133
step: 200, train loss1.4101, val loss: 1.3998
step: 300, train loss1.4342, val loss: 1.3892
step: 400, train loss1.4384, val loss: 1.4069
step: 500, train loss1.4516, val loss: 1.4127
step: 600, train loss1.4080, val loss: 1.4288
step: 700, train loss1.4450, val loss: 1.3907
step: 800, train loss1.4235, val loss: 1.3899
step: 900, train loss1.4003, val loss: 1.3567
```

```
Final loss: 1.3363757133483887
model saved
```

[35]:
```python
prompt = input("Prompt:\n")
context = torch.tensor(encode(prompt), dtype=torch.long, device=device)
generated_chars = decode(m.generate(context.unsqueeze(0),
  ↪max_new_tokens=150)[0].tolist())
print(f'Completion:\n{generated_chars}')
```

```
Prompt:
 the weather is
```

```
Completion:
the weather is to thing every emedite by sier's regulance of
Recorebuforenefoforesulaligorex Javevew
```

I

```
[ ]:
```