HW_13_GPT

April 19, 2023

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

# Hyperparameters.
# I suggest you start with very small values, unless you have a strong PC or␣
↪are running on the cluster                                                   ␣
↪
batch_size = 64 # How many independent sequences will we process in parallel? ␣
↪
block_size = 128 # What is the maximum context length for predictions?        ␣
↪
max_iters = 5000 # Max iterations we run the optimization
# How often we evaluate across the optimization; every 500 iterations
eval_interval = 500
learning_rate = 3e-4
"""
Use 'mps' if on a mac as below:

device = 'mps' if torch.backends.mps.is_available() else 'cpu'
"""
device = 'cuda'
# How many batches we use each time we evaluate
eval_iters = 200
# d_model = 96
# n_head = 6 # This implied that each head has a dimension for the key, query,␣
↪and values of d_model / 6.
# n_layer = 6 # This implies we have 6 turns to mix the embeddigs; this is "Nx"␣
↪in the paper
# dropout = 0.2
# ------------

d_model = 10
n_head = 2 # This implied that each head has a dimension for the key, query,␣
↪and values of d_model / 6.
n_layer = 2 # This implies we have 6 turns to mix the embeddigs; this is "Nx"␣
↪in the paper
```

```
dropout = 0.2


FILL_IN = "FILL_IN"

torch.manual_seed(1337)
```

[ ]: <torch._C.Generator at 0x27f547f9910>

```
[ ]: !gdown 'https://drive.google.com/uc?
      ↪export=download&id=1RlmRmXiWVKpZq98ftdtOIdM2lsA1uw3j'
```

```
'gdown' is not recognized as an internal or external command,
operable program or batch file.
'id' is not recognized as an internal or external command,
operable program or batch file.
```

As usual, we read the text file and then get two dictionaries from char to idx and in reverse. char embeddings is what we will use here.

```
[ ]: with open('hemingway.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)
     print(data.dtype)
     n = int(0.9*len(data)) # First 90% will be train, rest val                                 ␣
      ↪
     train_data = data[:n]
     val_data = data[n:]
```

```
torch.int64
```

[ ]:

```
# 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
    # Randomly select batch_size rows from data's row indices
    ix = torch.randint(0, data.shape[0]-block_size, (batch_size, ))
    # Select batch_size chuncks of text each of size block_size; stack them
    xb = torch.stack([data[i.item():i.item()+block_size] for i in ix])
    # Do the same for y, but make sure that this is shifted over by 1
    yb = torch.stack([data[i.item()+1:i.item()+block_size+1] for i in ix])
    # I.e. if you select xb (1, 2, 3, 4), yb should be (2, 3, 4, 5)
    xb, yb = xb.to(device), yb.to(device)
    # Each of xb, yb should be (batch_size, block_size)
    return xb, yb
```

[ ]:

```
```

[ ]:

```
@torch.no_grad()
def estimate_loss(model):
    out = {}
    # Put the model in eval mode here
    model.eval()
    for split in ['train', 'val']:
        losses = torch.zeros(eval_iters, device = device)
        for k in range(eval_iters):
            # Get a batch of data
            xb, yb = get_batch(split)
            # Get the mean and loss
            logits, loss = model(xb, yb)
            # Get the loss for this batch
            losses[k] = loss.item()
        # Insert the mean estimate for the loss, based on the slit you are in
        out[split] = loss.mean().item()
    # Put the model in train mode here
    model.train()
    return out
```

[ ]:

```
```

[ ]:

```
class Head(nn.Module):
    """
    This class represents one head of self-attention
    Note that since this is a Decoder, this is masked-self-attention
    There is no Encoder, so there is no cross-self-attention
    """
```

```
    def __init__(self, d_head):
        super().__init__()
        # Map each key, query, or value in to a d_head dimensional model.
        # Each should be matcies from d_model to d_head
        self.W_K = nn.Linear(d_model, d_head)
        self.W_Q = nn.Linear(d_model, d_head)
        self.W_V = nn.Linear(d_model, d_head)
        self.d_head = d_head
        self.register_buffer('tril', torch.tril(torch.ones(block_size,␣
↪block_size)))

        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        # (B, T, d_model)
        # B = batch_size, T = block_size in the below
        B,T,d = x.shape
        # Get the key and query representations from the embedding x

        # (B,T,d_head)
        k = self.W_K(x)
        # (B,T,d_head)                                                    ␣
↪
        q = self.W_Q(x)
        # (B,T,d_head)
        v = self.W_V(x)

        # Compute attention scores, and get the new representations for this␣
↪head

        # (B T, d_head) @ (B, d_head, T) = (B, T, T)
        # Multiply q by k and divide by the appropriate constant
        scores = torch.bmm(q, k.view(B, self.d_head, T))/torch.sqrt(torch.
↪tensor(self.d_head))

        # (B, T, T)
        # Apply a mask to scores, making all scores above the diagonal -inf  ␣
↪
        scores = scores.masked_fill(scores.tril()==0, torch.
↪tensor(float('-inf')))

        # (B, T, T)
        # Apply softmax to the final dimension of scores                      ␣
↪
        a =  torch.nn.Softmax(dim=-1)(scores)
```

```
        # Apply dropout                                                          ␣
↪
        a = self.dropout(a)
        # Perform the weighted aggregation of the values
        # Using a and v, get the new representations
        # (B, T, T) @ (B, T, d_head) -> (B, T, d_head)                           ␣
↪
        out = torch.bmm(a, v)
        # For each token, return the weighted sum of the values
        return out

class MultiHeadAttention(nn.Module):
    """
    Multiple heads of self-attention in parallel
    You can have just sequential code below
    """

    def __init__(self, num_heads, d_head):
        super().__init__()
        self.heads = nn.ModuleList([Head(d_head) for _ in range(num_heads)])
        # This is to project back to the dimension of d_model. In this case, it␣
↪is just a learned linear map
        self.W_O = torch.nn.Linear(num_heads*d_head, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        # Concatenate the different representations per head along the last␣
↪dimension
        out = torch.cat([head(x) for head in self.heads], dim=-1)
        # Project the concatenation and apply dropout; this is the W_O in␣
↪"Attention is all you need"
        out = self.dropout(self.W_O(out))
        return out
```

[ ]:
```
class FeedFoward(nn.Module):
    """
    A simple linear layer followed by a non-linearity; this is applied at the␣
↪token level
    """

    def __init__(self, d_model):
        super().__init__()
        d_ff = 4 * d_model
        # Map each token via a linear map to d_ff, apply ReLU, map back to␣
↪d_model, and then apply dropout
```

```
        # This can be done with nn.Sequential
        self.ff = torch.nn.Sequential(
            torch.nn.Linear(d_model, d_ff),
            torch.nn.ReLU(),
            torch.nn.Linear(d_ff, d_model),
            torch.nn.Dropout(dropout)
        )
    def forward(self, x):
        return self.ff(x)
```

[ ]:
```
class DecoderBlock(nn.Module):
    """
    Transformer decoder block: communication followed by computation
    These are stacked on top of each other one after another
    """

    def __init__(self, d_model, n_head):
        super().__init__()
        # Each head gets a smaller dimensional representation of the data
        # Assume each head gets a representation of dimension d_head and␣
↪d_model is divisible by n_head
        d_head = d_model // n_head
        self.sa = MultiHeadAttention(n_head, d_head)
        self.ff = FeedFoward(d_model)
        self.ln1 = nn.LayerNorm(d_model)
        self.ln2 = nn.LayerNorm(d_model)

    def forward(self, x):
        """
        This is different from the originl transformer paper
        In the "Attention is all you need" paper, we had
        x = self.ln1(x + self.sa(x))
        x = self.ln2(x + self.ffwd(x))
        See Figure 1 here, and mimic that: https://arxiv.org/pdf/2002.04745.pdf
        """
        x = x + self.sa(self.ln1(x))
        x = x + self.ff(self.ln2(x))
        return x
```

[ ]:
```
class GPT(nn.Module):
    def __init__(self):
        super().__init__()
        # Each token directly reads off the logits for the next token from a␣
↪lookup table
```

```
        # Token embeddings are from vocab_size to d_model
        self.token_embedding_table = torch.nn.Embedding(vocab_size, d_model)
        # Position embeddings are from block_size (T) to d_model
        self.position_embedding_table = torch.nn.Embedding(block_size, d_model)
        # This should be n_sequential applications of a DecoderBlock
        # This is the "Nx" piece in the paper
        self.blocks = torch.nn.Sequential(*[DecoderBlock(d_model, n_head) for _ in range(n_layer)])
         # Final layer norm
        self.ln = torch.nn.LayerNorm(d_model)
        self.ff = torch.nn.Linear(d_model, vocab_size)

    def forward(self, idx, targets=None):
        B, T = idx.shape

        # idx and targets are both (B,T) tensor of integers
        # (B,T,d_model)
        tok_emb = self.token_embedding_table(idx)
        # (T,d_model)
        pos_emb = self.position_embedding_table(idx)
        # Add positional encodings to encodings
        # (B,T,d_model)
        x = tok_emb + pos_emb

        # Mix up the token representations over and over via the blocks
        # (B,T,d_model)
        x = self.blocks(x)

        # Apply layer norm
        # (B,T,d_model)
        x = self.ln(x)

        # Apply the final linear map, to get to dimension vocab_size
        # (B,T,vocab_size)
        logits = self.ff(x)

        if targets is None:
            loss = None
        else:
            B, T, V = logits.shape
            logits = logits.view(B*T, V)
```

```python
            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
        This will generate B total paths in parrallel
        We will just geenrate 1 batch below
        """
        self.eval()
        for _ in range(max_new_tokens):
            # crop idx to the last block_size tokens
            # The model only has kowledge of the context of maximum size block_size
            # Get the newest (B, T) data; T = block_size
            idx_cond = idx[:, -block_size:]

            # Get the predictions
            # (B, T, vocab_size)
            logits, loss = self.forward(idx_cond)

            # Focus only on the last time step, get the logits
            # (B, vocab_size)
            logits = logits[:, -1, :]

            # Apply softmax to get probabilities
            # (B, vocab_size)
            probs = nn.functional.softmax(logits)

            # Sample from the distribution proporttional to probs
```

```
            # (B, 1)
↪
↪
↪
↪
↪
↪
            idx_next = torch.multinomial(probs, 1)

            # Append sampled index to the running sequence
            # (B, T+1)
↪
↪
↪
↪
↪
↪
            idx = torch.cat((idx, idx_next), 1)
        self.train()
        return idx
```

[ ]:
```
class EarlyStopping:
    def __init__(self, tolerance=5, min_delta=0):

        self.tolerance = tolerance
        self.min_delta = min_delta
        self.counter = 0
        self.early_stop = False

    def __call__(self, train_loss, validation_loss):
        if (validation_loss - train_loss) / train_loss > self.min_delta:
            self.counter += 1
            if self.counter >= self.tolerance:
                self.early_stop = True
```

[ ]:
```
model = GPT().to(device)
# Print the number of parameters in the model
↪
↪
↪
↪
↪
↪
print(sum(p.numel() for p in model.parameters()))
```

```
# Create a PyTorch optimizer
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, 1, gamma=0.9)
early_stopping = EarlyStopping(tolerance=1, min_delta=0.2)

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:
        if iter:
          scheduler.step()
        losses = estimate_loss(model)
        print(f"step {iter}: train loss {losses['train']:.4f}, val loss {losses['val']:.4f}")
        early_stopping(losses['train'], losses['val'])
        if early_stopping.early_stop:
          print("We stop at epoch {}".format(iter))
          break


    # 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()
```

```
5262
step 0: train loss 4.2127, val loss 4.2041
step 500: train loss 2.8627, val loss 2.8944
step 1000: train loss 2.5296, val loss 2.5746
step 1500: train loss 2.4317, val loss 2.4719
step 2000: train loss 2.3744, val loss 2.4223
step 2500: train loss 2.3610, val loss 2.3627
step 3000: train loss 2.3321, val loss 2.3800
step 3500: train loss 2.3021, val loss 2.3527
step 4000: train loss 2.2858, val loss 2.3385
step 4500: train loss 2.3163, val loss 2.3293
step 4999: train loss 2.3008, val loss 2.3143
```

[ ]:
```
# Start the model with a new line, generate up to 10000 tokens
# This is technically doing generations in batches, but here we have a batch␣
↪size of 1 and 1 element to start in the batch
# If you have a model that's very large, d_model = 384, n_head = 6, n_layer =␣
↪6, you'll get fairly decent results
context = torch.zeros((1, 1), dtype=torch.long, device=device)
print(decode(model.generate(context, max_new_tokens=100)[0].tolist()))
open('fake_hemingway.txt', 'w').write(decode(model.generate(context,␣
↪max_new_tokens=100)[0].tolist()))
```

```
C:\Users\Alex\AppData\Local\Temp\ipykernel_9944\1457601639.py:73: UserWarning:
Implicit dimension choice for softmax has been deprecated. Change the call to
include dim=X as an argument.
  probs = nn.functional.softmax(logits)



"exathenor til an het aseimed.

He re therstinsherd ma fat tricor t ayovine s."; the sh outhe srg T
```

[ ]: 101

[ ]:
```
torch.save(model.state_dict(), 'gpt.pt')
```