

cs2281r-pset0

Name: Kento Nishi

Assignment: pset 0

Repository: [KentoNishi/cs2281r-pset0](#)

Implementation Deviations

Parallelized Multi-Head Self-Attention

In the original tutorial, the multi-head self-attention mechanism is implemented in a concatenated list comprehension loop. This is quite inefficient, so I parallelized the computation using tensor operations. The code is based on my own fork of Andrej Karpathy's nanoGPT repository ([KentoNishi/generic-nanogpt](#)) which I created in 2023.

My `MultiHeadSelfAttention` class can be found on Line 94 ([permalink](#)).

Weight Tying between Token Embedding and LM Head

Weight tying is a common technique in transformer models to reduce the number of parameters. The original tutorial does not implement weight tying to keep the code simple, but I added it to my implementation.

The additional line of code can be found on Line 189 ([permalink](#)).

Saving the Model

For convenience, I saved the model and optimizer state dictionaries to a file named `model.pth`. The code can be found on Line 250 ([permalink](#)).

Output

Running `python gpt.py > output.txt` produces the following output:

`output.txt`

```
10.763969 M parameters
step 0: train loss 4.2058, val loss 4.2151
step 500: train loss 1.8398, val loss 1.9840
step 1000: train loss 1.4254, val loss 1.6283
step 1500: train loss 1.2813, val loss 1.5337
step 2000: train loss 1.2002, val loss 1.5086
step 2500: train loss 1.1329, val loss 1.4790
step 3000: train loss 1.0764, val loss 1.4879
step 3500: train loss 1.0224, val loss 1.4849
step 4000: train loss 0.9724, val loss 1.5081
```

step 4500: train loss 0.9186, val loss 1.5365
step 4999: train loss 0.8663, val loss 1.5676

BRUTUS:

O Mercutio, that any patience from the gentleman,
So much well breathe youth and to Juliet you?

First Citizen:

Fellow-man, ever women welcome of our own.
Will you do better? we will dispose to die,
repent the business to the driver
distrust of Rome.

Second Citizen:

He's enough.

MENENIUS:

Let the citizens get the sting; he is audacious
to whip something: advance him, an't human!

First Thian:

A noble city, a business case: as they speak the
with him about us.

CORIOLANUS:

Nay, sir

Code

gpt.py

```
1 import torch
2 import torch.nn as nn
3 from torch.nn import functional as F
4
5 batch_size = 64
6 block_size = 256
7 max_iters = 5000
8 eval_interval = 500
9 learning_rate = 3e-4
10 device = "cuda" if torch.cuda.is_available() else "cpu"
11 eval_iters = 200
12 n_embd = 384
13 n_head = 6
14 n_layer = 6
15 dropout = 0.2
```

```

16
17 torch.manual_seed(0)
18
19 with open("input.txt", "r", encoding="utf-8") as f:
20     text = f.read()
21
22 chars = sorted(list(set(text)))
23 vocab_size = len(chars)
24 stoi = {ch: i for i, ch in enumerate(chars)}
25 itos = {i: ch for i, ch in enumerate(chars)}
26 encode = lambda s: [stoi[c] for c in s]
27 decode = lambda l: "".join([itos[i] for i in l])
28
29 data = torch.tensor(encode(text), dtype=torch.long)
30 n = int(0.9 * len(data))
31 train_data = data[:n]
32 val_data = data[n:]
33
34
35 def get_batch(split):
36     data = train_data if split == "train" else val_data
37     ix = torch.randint(len(data) - block_size, (batch_size,))
38     x = torch.stack([data[i : i + block_size] for i in ix])
39     y = torch.stack([data[i + 1 : i + block_size + 1] for i in ix])
40     x, y = x.to(device), y.to(device)
41     return x, y
42
43
44 @torch.no_grad()
45 def estimate_loss():
46     out = {}
47     model.eval()
48     for split in ["train", "val"]:
49         losses = torch.zeros(eval_iters)
50         for k in range(eval_iters):
51             X, Y = get_batch(split)
52             logits, loss = model(X, Y)
53             losses[k] = loss.item()
54         out[split] = losses.mean()
55     model.train()
56     return out
57
58
59 # class Head(nn.Module):
60 #     def __init__(self, head_size):
61 #         super().__init__()

```

```

62 #         self.key = nn.Linear(n_embd, head_size, bias=False)
63 #         self.query = nn.Linear(n_embd, head_size, bias=False)
64 #         self.value = nn.Linear(n_embd, head_size, bias=False)
65 #         self.register_buffer("tril", torch.tril(torch.ones(block_size, block_size)))
66 #         self.dropout = nn.Dropout(dropout)
67
68 #     def forward(self, x):
69 #         B, T, C = x.shape
70 #         k = self.key(x)
71 #         q = self.query(x)
72 #         wei = q @ k.transpose(-2, -1) * k.shape[-1] ** -0.5
73 #         wei = wei.masked_fill(self.tril[:T, :T] == 0, float("-inf"))
74 #         wei = F.softmax(wei, dim=-1)
75 #         wei = self.dropout(wei)
76 #         v = self.value(x)
77 #         out = wei @ v
78 #         return out
79
80
81 # class MultiHeadAttention(nn.Module):
82 #     def __init__(self, num_heads, head_size):
83 #         super().__init__()
84 #         self.heads = nn.ModuleList([Head(head_size) for _ in range(num_heads)])
85 #         self.proj = nn.Linear(head_size * num_heads, n_embd)
86 #         self.dropout = nn.Dropout(dropout)
87
88 #     def forward(self, x):
89 #         out = torch.cat([h(x) for h in self.heads], dim=-1)
90 #         out = self.dropout(self.proj(out))
91 #         return out
92
93
94 # CUSTOM: Parallelized Multi-Head Self-Attention
95 class MultiHeadSelfAttention(nn.Module):
96     def __init__(
97         self, num_heads, head_size, n_embd, dropout=dropout, block_size=block_size
98     ):
99         super().__init__()
100         self.num_heads = num_heads
101         self.head_size = head_size
102         self.n_embd = n_embd
103
104         self.key = nn.Linear(n_embd, num_heads * head_size, bias=False)
105         self.query = nn.Linear(n_embd, num_heads * head_size, bias=False)
106         self.value = nn.Linear(n_embd, num_heads * head_size, bias=False)
107         self.proj = nn.Linear(num_heads * head_size, n_embd)

```

```

1108         self.dropout = nn.Dropout(dropout)
1109
1110         self.register_buffer("tril", torch.tril(torch.ones(block_size, block_size)))
1111
1112     def forward(self, x):
1113         B, T, C = x.shape
1114
1115         k = self.key(x).view(
1116             B, T, self.num_heads, self.head_size
1117         ) # (B, T, num_heads, head_size)
1118         q = self.query(x).view(
1119             B, T, self.num_heads, self.head_size
1120         ) # (B, T, num_heads, head_size)
1121         v = self.value(x).view(
1122             B, T, self.num_heads, self.head_size
1123         ) # (B, T, num_heads, head_size)
1124
1125         k = k.transpose(1, 2) # (B, num_heads, T, head_size)
1126         q = q.transpose(1, 2) # (B, num_heads, T, head_size)
1127         v = v.transpose(1, 2) # (B, num_heads, T, head_size)
1128
1129         wei = q @ k.transpose(-2, -1) * self.head_size**-0.5 # (B, num_heads, T, T)
1130         wei = wei.masked_fill(
1131             self.tril[:T, :T] == 0, float("-inf"))
1132         ) # (B, num_heads, T, T)
1133         wei = F.softmax(wei, dim=-1) # (B, num_heads, T, T)
1134         wei = self.dropout(wei)
1135
1136         out = (
1137             wei @ v
1138         ) # (B, num_heads, T, T) @ (B, num_heads, T, head_size) --> (B, num_heads, T, head_size)
1139
1140         out = (
1141             out.transpose(1, 2).contiguous().view(B, T, self.num_heads * self.head_size)
1142         )
1143
1144         out = self.dropout(self.proj(out))
1145         return out
1146
1147
1148     class FeedFoward(nn.Module):
1149         def __init__(self, n_embd):
1150             super().__init__()
1151             self.net = nn.Sequential(
1152                 nn.Linear(n_embd, 4 * n_embd),
1153                 nn.ReLU(),

```

```

154         nn.Linear(4 * n_embd, n_embd),
155         nn.Dropout(dropout),
156     )
157
158     def forward(self, x):
159         return self.net(x)
160
161
162     class Block(nn.Module):
163         def __init__(self, n_embd, n_head):
164             super().__init__()
165             head_size = n_embd // n_head
166             # self.sa = MultiHeadAttention(n_head, head_size)
167             self.sa = MultiHeadSelfAttention(n_head, head_size, n_embd)
168             self.ffwd = FeedFoward(n_embd)
169             self.ln1 = nn.LayerNorm(n_embd)
170             self.ln2 = nn.LayerNorm(n_embd)
171
172         def forward(self, x):
173             x = x + self.sa(self.ln1(x))
174             x = x + self.ffwd(self.ln2(x))
175             return x
176
177
178     class GPTLanguageModel(nn.Module):
179         def __init__(self):
180             super().__init__()
181             self.token_embedding_table = nn.Embedding(vocab_size, n_embd)
182             self.position_embedding_table = nn.Embedding(block_size, n_embd)
183             self.blocks = nn.Sequential(
184                 *[Block(n_embd, n_head=n_head) for _ in range(n_layer)]
185             )
186             self.ln_f = nn.LayerNorm(n_embd)
187             self.lm_head = nn.Linear(n_embd, vocab_size)
188
189             # CUSTOM: Weight Tying between Token Embedding and LM Head
190             self.lm_head.weight = self.token_embedding_table.weight
191
192             self.apply(self._init_weights)
193
194         def _init_weights(self, module):
195             if isinstance(module, nn.Linear):
196                 torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
197                 if module.bias is not None:
198                     torch.nn.init.zeros_(module.bias)
199             elif isinstance(module, nn.Embedding):

```

```

200         torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
201
202     def forward(self, idx, targets=None):
203         B, T = idx.shape
204         tok_emb = self.token_embedding_table(idx)
205         pos_emb = self.position_embedding_table(torch.arange(T, device=device))
206         x = tok_emb + pos_emb
207         x = self.blocks(x)
208         x = self.ln_f(x)
209         logits = self.lm_head(x)
210         if targets is None:
211             loss = None
212         else:
213             B, T, C = logits.shape
214             logits = logits.view(B * T, C)
215             targets = targets.view(B * T)
216             loss = F.cross_entropy(logits, targets)
217         return logits, loss
218
219     def generate(self, idx, max_new_tokens):
220         for _ in range(max_new_tokens):
221             idx_cond = idx[:, -block_size:]
222             logits, loss = self(idx_cond)
223             logits = logits[:, -1, :]
224             probs = F.softmax(logits, dim=-1)
225             idx_next = torch.multinomial(probs, num_samples=1)
226             idx = torch.cat((idx, idx_next), dim=1)
227         return idx
228
229
230 model = GPTLanguageModel()
231 m = model.to(device)
232 print(sum(p.numel() for p in m.parameters()) / 1e6, "M parameters")
233
234 optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)
235 for iter in range(max_iters):
236     if iter % eval_interval == 0 or iter == max_iters - 1:
237         losses = estimate_loss()
238         print(
239             f"step {iter}: train loss {losses['train']:.4f}, val loss {losses['val']:.4f}"
240         )
241     xb, yb = get_batch("train")
242     logits, loss = model(xb, yb)
243     optimizer.zero_grad(set_to_none=True)
244     loss.backward()
245     optimizer.step()

```

```
246
247 context = torch.zeros((1, 1), dtype=torch.long, device=device)
248 print(decode(m.generate(context, max_new_tokens=500)[0].tolist()))
249
250 # CUSTOM: Save model and optimizer state dicts
251 torch.save(
252     {"model": model.state_dict(), "optimizer": optimizer.state_dict()}, "model.pth"
253 )
```