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 190 (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 jumet you?

First Citizen:
Fellow-man, e'er wemen welcong of our own.
Will you do retur guiter'd? we will dispose to die,
repent the business to the driver
distrustipe of Rome.

Second Citizen:
He's enough.

MENENIUS:
Let the citus got he sting; he is audired
to whip something: advance him, an't human!

First Thizan:
A noble city, a business ca: as they speak the
withing him about us.

CORIOLANUS:
Nay, si
```

Code

gpt.py

```python
import torch
import torch.nn as nn
from torch.nn import functional as F

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

```python

torch.manual_seed(0)

with open("input.txt", "r", encoding="utf-8") as f:
    text = f.read()

chars = sorted(list(set(text)))
vocab_size = len(chars)
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]
decode = lambda l: "".join([itos[i] for i in l])

data = torch.tensor(encode(text), dtype=torch.long)
n = int(0.9 * len(data))
train_data = data[:n]
val_data = data[n:]


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


@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


# 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
#         k = self.key(x)
#         q = self.query(x)
#         wei = q @ k.transpose(-2, -1) * k.shape[-1] ** -0.5
#         wei = wei.masked_fill(self.tril[:T, :T] == 0, float("-inf"))
#         wei = F.softmax(wei, dim=-1)
#         wei = self.dropout(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(head_size * num_heads, 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


# CUSTOM: Parallelized Multi-Head Self-Attention
class MultiHeadSelfAttention(nn.Module):
    def __init__(
        self, num_heads, head_size, n_embd, dropout=dropout, block_size=block_size
    ):
        super().__init__()
        self.num_heads = num_heads
        self.head_size = head_size
        self.n_embd = n_embd

        self.key = nn.Linear(n_embd, num_heads * head_size, bias=False)
        self.query = nn.Linear(n_embd, num_heads * head_size, bias=False)
        self.value = nn.Linear(n_embd, num_heads * head_size, bias=False)
        self.proj = nn.Linear(num_heads * head_size, n_embd)
```

```python
        self.dropout = nn.Dropout(dropout)

        self.register_buffer("tril", torch.tril(torch.ones(block_size, block_size)))

    def forward(self, x):
        B, T, C = x.shape

        k = self.key(x).view(
            B, T, self.num_heads, self.head_size
        )  # (B, T, num_heads, head_size)
        q = self.query(x).view(
            B, T, self.num_heads, self.head_size
        )  # (B, T, num_heads, head_size)
        v = self.value(x).view(
            B, T, self.num_heads, self.head_size
        )  # (B, T, num_heads, head_size)

        k = k.transpose(1, 2)  # (B, num_heads, T, head_size)
        q = q.transpose(1, 2)  # (B, num_heads, T, head_size)
        v = v.transpose(1, 2)  # (B, num_heads, T, head_size)

        wei = q @ k.transpose(-2, -1) * self.head_size**-0.5  # (B, num_heads, T, T)
        wei = wei.masked_fill(
            self.tril[:T, :T] == 0, float("-inf")
        )  # (B, num_heads, T, T)
        wei = F.softmax(wei, dim=-1)  # (B, num_heads, T, T)
        wei = self.dropout(wei)

        out = (
            wei @ v
        )  # (B, num_heads, T, T) @ (B, num_heads, T, head_size) --> (B, num_heads, T, head_

        out = (
            out.transpose(1, 2).contiguous().view(B, T, self.num_heads * self.head_size)
        )

        out = self.dropout(self.proj(out))
        return out


class FeedFoward(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):
    def __init__(self, n_embd, n_head):
        super().__init__()
        head_size = n_embd // n_head
        # self.sa = MultiHeadAttention(n_head, head_size)
        self.sa = MultiHeadSelfAttention(n_head, head_size, n_embd)
        self.ffwd = FeedFoward(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.ffwd(self.ln2(x))
        return x


class GPTLanguageModel(nn.Module):
    def __init__(self):
        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)
        self.lm_head = nn.Linear(n_embd, vocab_size)

        # CUSTOM: Weight Tying between Token Embedding and LM Head
        self.lm_head.weight = self.token_embedding_table.weight

        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):
```

```python
                torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)

    def forward(self, idx, targets=None):
        B, T = idx.shape
        tok_emb = self.token_embedding_table(idx)
        pos_emb = self.position_embedding_table(torch.arange(T, device=device))
        x = tok_emb + pos_emb
        x = self.blocks(x)
        x = self.ln_f(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):
        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 = GPTLanguageModel()
m = model.to(device)
print(sum(p.numel() for p in m.parameters()) / 1e6, "M parameters")

optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)
for iter in range(max_iters):
    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}"
        )
    xb, yb = get_batch("train")
    logits, loss = model(xb, yb)
    optimizer.zero_grad(set_to_none=True)
    loss.backward()
    optimizer.step()
```

```python

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

# CUSTOM: Save model and optimizer state dicts
torch.save(
    {"model": model.state_dict(), "optimizer": optimizer.state_dict()}, "model.pth"
)
```