Phase 3: LLaMA2 Model Training

Implementations

1. Initial Setup

For intinial testing, the decoder layer is set to be only one.

```
model_args.n_layers = 1  # for debugging purposes we only use 1 layer
```

In the `model.py` file, remove all the `start_pos` variables in forward functions and replace them with 0. This is because we removed the KV-caching and `start_pos` is no longer needed.

Besides, I changed the max sequence length so that it can fit for the Alpaca data:

```
max_seq_len: int = 512      # modify to a larger max_seq_len
```

2. End-to-End Instruction Tuning Flow

Code was provided with the assignment. The classes included `SupervisedDataset(Dataset)` and `DataCollatorForSupervisedDataset`. The main part our modification focused on is the `train()` function.

I also add helper functions `start_timer()` and `end_timer_and_print()` to record the runtime and memory usage.

```
def start_timer():
    global start_time
    gc.collect()
    torch.cuda.empty_cache()
    torch.cuda.reset_max_memory_allocated()
    torch.cuda.synchronize()
    start_time = time.time()


def end_timer_and_print(local_msg=""):
    torch.cuda.synchronize()
    end_time = time.time()
    print("\n" + local_msg)
    print("Total runtime = {:.3f} sec".format(end_time - start_time))
    print("Peak memory usage = {} MBytes".format(torch.cuda.max_memory_allocated()/1000))
```

3. Training Iteration Loop

Code was provided. The crucial change to replace HuggingFace's object with Alpaca's repo is inside the `train()` function. After assigning the paths of our model and data, we extracted the labels and logits within the bath according to the format of Alpaca data.

4. Gradient Accumulation and Mixed Precision Training

For mixed precision, I followed the instructions of AMP Example. `GradScaler` was utilized to prevent gradient vanishing. `Autocast` set the demanding `dtype` to be `torch.float16` for the cuda during training.

Changes are made in the `finetuning.py` file:

```
scaler = GradScaler()   # scale the gradient

    for epoch in range(5):
        for i, batch in enumerate(dataloader):
            input_ids = batch['input_ids'].to("cuda")
            labels = batch['labels'].to("cuda")

            # auto mixed precision
            with autocast(dtype=torch.float16):
                logits = model(input_ids)

                shift_logits = logits[..., :-1, :].contiguous()
                shift_labels = labels[..., 1:].contiguous()
                shift_logits = shift_logits.view(-1, 32000)
                shift_labels = shift_labels.view(-1)

                loss = criterion(shift_logits, shift_labels)

            # exit autocast before backward()
            scaler.scale(loss).backward()
            scaler.step(optimizer)
            scaler.update()
            optimizer.zero_grad()
```

Then, I added the Gradient Accumulation, and assigned `accumulation_steps` to be 8. The gradient is only accumulated and backwarded when reaching this amount of steps:

```
    accumulation_steps = 8
    scaler = GradScaler()

    for epoch in range(5):
        for i, batch in enumerate(dataloader):
            input_ids = batch['input_ids'].to("cuda")
            labels = batch['labels'].to("cuda")
```

```
            # auto mixed precision
            with autocast(dtype=torch.float16):
                logits = model(input_ids)

                shift_logits = logits[..., :-1, :].contiguous()
                shift_labels = labels[..., 1:].contiguous()
                shift_logits = shift_logits.view(-1, 32000)
                shift_labels = shift_labels.view(-1)

                # loss should be devided by accumulation_steps
                loss = criterion(shift_logits, shift_labels) / accumulation_steps

            # exit autocast before backward()
            scaler.scale(loss).backward()

            # accumulate gradient
            if (i + 1) % accumulation_steps == 0:
                scaler.step(optimizer)
                scaler.update()
                optimizer.zero_grad()
```

5. LoRA Linear Layer Module

The file `lora.py` was created based on the official instructions. I copied the classes `LoRALayer()` and `Linear(nn.Linear, LoRALayer)` from the official code, so that I can convert the model using RoLA Linear modules.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

class LoRALayer():
    def __init__():
        ...

class Linear(nn.Linear, LoRALayer):
    def __init__():
        ...
    def reset_parameters(self):
        ...
    def train(self, mode: bool = True):
        ...
    def forward(self, x: torch.Tensor):
        ...
```

Then in the `model.py` file, I modified the way Q and V are projected in the `Attention` class:

```
from llama.lora import Linear as LoRALinear

...

class Attention(nn.Module):
    """Multi-head attention module."""
    def __init__(self, args: ModelArgs):

        ...

        # LoRA Linear
        self.wq = LoRALinear(args.dim, args.n_heads * self.head_dim,
                             merge_weights=False, bias=False)
        self.wv = LoRALinear(args.dim, self.n_kv_heads * self.head_dim,
                             merge_weights=False, bias=False)

        # origin code
        # self.wq = nn.Linear(args.dim, args.n_heads * self.head_dim, bias=False)
        # self.wv = nn.Linear(args.dim, self.n_kv_heads * self.head_dim, bias=False)

        self.wk = nn.Linear(args.dim, self.n_kv_heads * self.head_dim, bias=False)
        self.wo = nn.Linear(args.n_heads * self.head_dim, args.dim, bias=False)
```

Also in the `finetuning.py`, de-comment the part related to RoLA weights:

```
    # Freeze model parameters other than lora weights
    for name, params in model.named_parameters():
        if "lora_" in name:
            params.requires_grad = True
        else:
            params.requires_grad = False
```

6. Gradient Checkpointing

Checkpoints were inserted in the `model.py` file. After importing the `checkpoint` in Pytorch, I changed the `forward` function in `TransformerBlock`. I chose to apply the gradient checkpoint for every training layer. Custom attention and feed_forward functions were created in order to separate them from the checkpoint processing:

```
def forward(
    self,
    x: torch.Tensor,
    freqs_cis: torch.Tensor,
    mask: Optional[torch.Tensor],
```

```
):
    def custom_attention(x, freqs_cis, mask):
        return x + self.attention(self.attention_norm(x), freqs_cis, mask)

    def custom_feed_forward(h):
        return h + self.feed_forward(self.ffn_norm(h))

    if self.training:
        h = checkpoint(custom_attention, x, freqs_cis, mask, use_reentrant=False)
        out = checkpoint(custom_feed_forward, h, use_reentrant=False)
    else:
        h = custom_attention(x, freqs_cis, mask)
        out = custom_feed_forward(h)

    return out
```

7. Model Fine-Tuning

The GPU type was switched to A100; the dataset path was changed to the Alpaca 200 samples:

```
data_path = "/project/saifhash_1190/llama2-7b/alpaca_data_200.json"
```

8. Hyperparameters

Set the variables according to the instruction:

```
# set the learning rate to be 1e-5
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-5)
...
# set the batch size to be 1
dataloader = torch.utils.data.DataLoader(
        data_module["train_dataset"],
        batch_size=1,
        collate_fn=data_module["data_collator"],
        shuffle=True,
    )
...
# set the gradient accumulation step to be 8
accumulation_steps = 8
```

For LoRA configuration:

```
# set r = 16, alpha = 32, and dropout rate = 0.05
class Linear(nn.Linear, LoRALayer):
    def __init__(
            self,
            in_features: int,
            out_features: int,
            r: int = 16,
            lora_alpha: int = 32,
            lora_dropout: float = 0.05,
            merge_weights: bool = True,
            **kwargs
    ):
    ...
```

Table 1

		Grad. Accumulation	Grad. Checkpoint	Mixed Precision	LoRA
Memory	parameter	↓	↓	↓	↓
	activation	-	↓	↓	↓
	gradient	-	-	↓	-
	optimizer state	↓	-	-	-
Computation		↓	↑	↓	↓

Table 2

While training with 32 layers, techniques without LoRA Linear Layer will all result in an "out of memory error". The table is shown below:

(num of Layers: 32)

GA	OFF				ON			
MP	OFF		ON		OFF		ON	
LoRA	OFF	ON	OFF	ON	OFF	ON	OFF	ON
Peak Mem	-	3.13×10^7	-	4.34×10^7	-	3.13×10^7	-	4.34×10^7
Runtime	-	256.750	-	90.911	-	253.861	-	90.630

As we can see in this table, the Mixed Precision technique results in a higher peak memory consumption, but saves largely in the runtime. In the meanwhile, Gradient Accumulation doesn't contribute much to the improvement of the training.

To see what is the scenario when LoRA is off, I changed the layer to 8 while turning off the LoRA Linear Layer. The table is shown below:

(num of Layers: 8)

GA	OFF		ON	
MP	OFF	ON	OFF	ON
LoRA	OFF			
Peak Mem	3.77×10^7	2.02×10^7	3.77×10^7	2.02×10^7
Runtime	214.698	68.415	125.985	56.547

Without the LoRA Linear Layer, the effect of Mixed Precision is similar to before, but we can also observe that the Gradient Accumulation, in this case, could make the runtime slower, especially in the situation where the runtime is high.