Phase 3

Implementation Details

1. Changes made to finetuning.py

a. We have made changes to the finetuning.py module to handle the requires_grad parameter depending on the implementation of lora.

```
print(model)
for name, params in model.named_parameters():
    if "lora_" in name:
        params.requires_grad = True
    else:
        params.requires_grad = False
```

b. We made additional changes to implement Mixed precision using the autocast() function.

```
    for step, batch in enumerate(dataloader):
        with autocast():  # Enable automatic mixed precision
            input_ids = batch['input_ids'].to("cuda")
            labels = batch['labels'].to("cuda")
            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) / accumulation_steps  # Scale the loss by accumulation steps

        #Implementation without Mixed Precision
        # input_ids = batch['input_ids'].to("cuda")
        # labels = batch['labels'].to("cuda")
        # try:
        #     logits = model(input_ids)
        # except:
        #     continue
        # 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) / accumulation_steps  # Scale the loss by accumulation steps


        scaler.scale(loss).backward()  # Scale the loss and call backward to accumulate gradients
```

c. We implemented gradient accumulation using the scaler class and pushed them into the lora_weights file to be later used for inference.

```python
        scaler.scale(loss).backward()  # Scale the loss and call backward to accumulate gradients

        if (step + 1) % accumulation_steps == 0 or step == len(dataloader)-1:
            scaler.step(optimizer)  # Perform a parameter update
            scaler.update()  # Update the scale for next iteration
            optimizer.zero_grad()  # Zero the parameter gradients

            model_weights = model.state_dict()
            lora_weights = {k: v for k, v in model_weights.items() if "lora_" in k}
            torch.save(lora_weights, "lora_weights.pth")

        if step % 50 == 0:
            print(f"Epoch: {epoch} | Iter: {step} | Loss: {loss.item() * accumulation_steps}")
            #print(loss.item() * accumulation_steps)  # Print the unscaled loss

#Printing Memory Required
peak_memory_allocated_bytes = torch.cuda.max_memory_allocated()
peak_memory_allocated_gb = peak_memory_allocated_bytes / (1024 ** 2)
print(f"Peak memory allocated: {peak_memory_allocated_gb:.2f} MB")
```

d. We utilized the time module to measure the execution duration of the program.

```python
if __name__ == "__main__":
    start_time = time.time()
    train()

    # Record the end time
    end_time = time.time()

    # Calculate and print the elapsed time
    elapsed_time = end_time - start_time
    print(f"Time taken: {elapsed_time} seconds")
```

2. lora.py

The Linear class extends PyTorch's nn.Linear module to implement the LoRA mechanism. It introduces parameters for LoRA, such as the dimensionality of the random matrix (r), a scaling factor (lora_alpha), and an optional dropout rate (lora_dropout).

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

import math
from typing import Optional, List

class LoRALayer():
    def __init__(
        self,
        r: int,
        lora_alpha: int,
        lora_dropout: float,
        merge_weights: bool,
    ):
        self.r = r
        self.lora_alpha = lora_alpha
        # Optional dropout
        if lora_dropout > 0.:
            self.lora_dropout = nn.Dropout(p=lora_dropout)
        else:
            self.lora_dropout = lambda x: x
        # Mark the weight as unmerged
        self.merged = False
        self.merge_weights = merge_weights
```

```python
class Linear(nn.Linear, LoRALayer):
    # LoRA implemented in a dense layer
    def __init__(
        self,
        in_features: int,
        out_features: int,
        r: int = 0,
        lora_alpha: int = 1,
        lora_dropout: float = 0.,
        fan_in_fan_out: bool = False, # Set this to True if the layer to replace stores weight like (f
        merge_weights: bool = True,
        **kwargs
    ):
        nn.Linear.__init__(self, in_features, out_features, **kwargs)
        LoRALayer.__init__(self, r=r, lora_alpha=lora_alpha, lora_dropout=lora_dropout,
                           merge_weights=merge_weights)

        self.fan_in_fan_out = fan_in_fan_out
        # Actual trainable parameters
        if r > 0:
            self.lora_A = nn.Parameter(self.weight.new_zeros((r, in_features)))
            self.lora_B = nn.Parameter(self.weight.new_zeros((out_features, r)))

            self.scaling = self.lora_alpha / self.r
            # Freezing the pre-trained weight matrix
            self.weight.requires_grad = False
```

```python
        self.reset_parameters()
        if fan_in_fan_out:
            self.weight.data = self.weight.data.transpose(0, 1)

    def reset_parameters(self):
        nn.Linear.reset_parameters(self)
        if hasattr(self, 'lora_A'):
            # initialize A the same way as the default for nn.Linear and B to zero
            nn.init.kaiming_uniform_(self.lora_A, a=math.sqrt(5))
            nn.init.zeros_(self.lora_B)

    def train(self, mode: bool = True):
        def T(w):
            return w.transpose(0, 1) if self.fan_in_fan_out else w
        nn.Linear.train(self, mode)
        if mode:
            if self.merge_weights and self.merged:
                # Make sure that the weights are not merged
                if self.r > 0:
                    self.weight.data -= T(self.lora_B @ self.lora_A) * self.scaling
                self.merged = False
        else:
            if self.merge_weights and not self.merged:
                # Merge the weig (function) data: Any
                if self.r > 0:
                    self.weight.data += T(self.lora_B @ self.lora_A) * self.scaling
                self.merged = True
```

```
f forward(self, x: torch.Tensor):
  def T(w):
      return w.transpose(0, 1) if self.fan_in_fan_out else w
  if self.r > 0 and not self.merged:
      result = F.linear(x, T(self.weight), bias=self.bias)
      result += (self.lora_dropout(x) @ self.lora_A.transpose(0, 1) @ self.lora_B.transpose(0, 1)) * self.scaling
      return result
  else:
      return F.linear(x, T(self.weight), bias=self.bias)
  print(self.r, self.lora_alpha)
```

3. Changes made to model.py

In the below code we called the Linear class to implement lora

```
#implemented lora
#self.wq = nn.Linear(args.dim, args.n_heads * self.head_dim, bias=False)
self.wq = Linear(args.dim, args.n_heads * self.head_dim, r = 32, lora_alpha = 64, lora_dropout = 0.05, bias=False)
self.wk = nn.Linear(args.dim, self.n_kv_heads * self.head_dim, bias=False)
#self.wv = nn.Linear(args.dim, self.n_kv_heads * self.head_dim, bias=False)
self.wv = Linear(args.dim, self.n_kv_heads * self.head_dim, r = 32, lora_alpha = 64, lora_dropout = 0.05, bias=False)
self.wo = nn.Linear(args.n_heads * self.head_dim, args.dim, bias=False)
peak_memory_allocated_bytes = torch.cuda.max_memory_allocated()
peak_memory_allocated_gb = peak_memory_allocated_bytes / (1024 ** 2)
#print(f"Peak memory allocated: {torch.cuda.max_memory_allocated()}")
```

The below code iterates through the layers of the neural network models, applying checkpointing to alternate transformer blocks

```
# #code without gradient checkpointing
# for layer in self.layers:
#     h = layer(h, 0 , freqs_cis, mask)

#code with gradient checkpointing
for i, layer in enumerate(self.layers):
    if (i+1) % 2 == 0:
        # Apply checkpointing to every alternate transformer block
        h = checkpoint(layer, h, 0, freqs_cis, mask, use_reentrant=False)
    else:
        h = layer(h, 0, freqs_cis, mask)
```

4. Changes made to inference.py

a. Added code to accumulate the LoRA weights loaded from the file lora_weights to the model's current state dictionary model_weights.

```python
# Load the saved LoRA weights
lora_weights = torch.load(lora_weights_path, map_location="cuda")

# Update the model's state_dict with the loaded LoRA weights
model_weights = model.state_dict()
for k, v in lora_weights.items():
    model_weights[k] += v

# Load the updated state_dict into the model
model.load_state_dict(model_weights)

prompts = [
    # For these prompts, the expected answer is the natural continuation of the prompt
    "I believe the meaning of life is",
    "Simply put, the theory of relativity states that ",
    """A brief message congratulating the team on the launch:
```

b. Also added additional prompts from the Alpaca Dataset.

```python
prompts = [
    # For these prompts, the expected answer is the natural continuation of the prompt
    "I believe the meaning of life is",
    "Simply put, the theory of relativity states that ",
    """A brief message congratulating the team on the launch:

    Hi everyone,

    I just """,
    # Few shot prompt (providing a few examples before asking model to complete more);
    """Translate English to French:

    sea otter => loutre de mer
    peppermint => menthe poivrée
    plush girafe => girafe peluche
    cheese =>""",

    """Describe the structure of an atom.""",

    """Rewrite the following sentence using active voice.
        The news report was read by the captain.
    """
]
```

Results

1. Inference output

```
[sumukhsh@b01-01 ml-systems-final-project-khatrikhushboo1]$ python inference.py
/home1/sumukhsh/.local/lib/python3.9/site-packages/torch/__init__.py:696: UserWarning: torch.set_default_tensor_type() is deprecated as of PyTorch 2.1, please use torch.set_default_dt
ype() and torch.set_default_device() as alternatives. (Triggered internally at ../torch/csrc/tensor/python_tensor.cpp:451.)
  _C._set_default_tensor_type(t)
I believe the meaning of life is
> to live it and enjoy every moment. It is to find happiness in the small things and to appreciate every day. It is to make the most of our time and to leave a positive impact on the
world.

==================================

Simply put, the theory of relativity states that
> 1) the speed of light is the same for all observers, regardless of their motion relative to the light source, and 2) the laws of physics are the same for all observers, regardless o
f their motion relative to each other.
The theory of relativity is a major departure from the classical physics of

==================================

A brief message congratulating the team on the launch:

        Hi everyone,

        I just
> wanted to say a huge congratulations to the team on the launch of the website. It looks fantastic and I'm sure it will be a huge success. I can't wait to see what the future holds f
or the site!

==================================

Translate English to French:

        sea otter => loutre de mer
        peppermint => menthe poivrée
        plush girafe => girafe peluche
        cheese =>
> fromage
        cat => chat
        dog => chien
        mouse => souris
        catapult => catapulte
        potato => pomme de terre
```

```
> fromage
        cat => chat
        dog => chien
        mouse => souris
        catapult => catapulte
        potato => pomme de terre
        pizza => pizza
        hamburger => hamburger
        chocolate => chocol

==================================

Describe the structure of an atom.
> The nucleus consists of protons and neutrons, which are bound by the force of the strong nuclear force. The protons have a positive charge, while the neutrons have no charge. The nu
mber of each particle determines the atomic number of the atom and the type of element.
What are the 5

==================================

Rewrite the following sentence using active voice.
                The news report was read by the captain.

>       The captain read the news report.

==================================
```

2. Table 1 and Table 2:

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

Table 1: System performance analysis

GA	OFF				ON			
MP	OFF		ON		OFF		ON	
LoRA	OFF	ON	OFF	ON	OFF	ON	OFF	ON
Peak Mem	X	27131.95	X	33217.96	X	27163.95	X	33249.97
Runtime	X	328.49	X	175.99	X	326.97	X	192.98

Table 2: System performance measurement