ROHAN VERMA 2K20/CE/129

TASK1

```
# Install the transformers library
# pip install transformers

from transformers import GPT2Tokenizer, GPT2LMHeadModel
import torch

# Load pre-trained GPT-2 model and tokenizer
model_name = "gpt2"  # You can also use "gpt2-medium", "gpt2-large", "gpt2-xl" for larger
models
tokenizer = GPT2Tokenizer.from_pretrained(model_name)
model = GPT2LMHeadModel.from_pretrained(model_name)

def generate_text(prompt, max_length=100, temperature=1.0):
    # Tokenize the input text
    input_ids = tokenizer.encode(prompt, return_tensors="pt")

    # Generate text using the GPT-2 model
    output = model.generate(input_ids, max_length=max_length, temperature=temperature,
num_return_sequences=1)

    # Decode the generated text
    generated_text = tokenizer.decode(output[0], skip_special_tokens=True)

    return generated_text

# Demonstrate GPT-2 text generation
prompt = "Once upon a time"
generated_story = generate_text(prompt)
print("Generated Story:")
print(generated_story)

# Testing to verify functioning
assert len(generated_story) > 0, "The generated story is empty."

print("Testing Passed!")
```

TASK2

1. Rotary Positional Embedding:

Rotary Positional Embedding is a technique introduced in RoFormer. To replace the original positional embeddings in GPT-2 with rotary embeddings, you would need to modify the architecture of the model. The rotary embeddings are designed to capture sequential information more effectively. Here's a simplified example of how you might modify the GPT-2 model using Rotary Positional Embedding:

```
from transformers import GPT2Model, GPT2Config
import torch
import torch.nn.functional as F

class GPT2WithRotaryEmbedding(GPT2Model):
    def __init__(self, config):
        super().__init__(config)
        # Add rotary positional embeddings here

    def forward(self, input_ids=None, attention_mask=None, **kwargs):
        # Modify the forward pass to incorporate rotary positional embeddings
        # ...
        return super().forward(input_ids, attention_mask=attention_mask, **kwargs)
```

. Group Query Attention:

Group Query Attention is a mechanism introduced in the GQA paper. It involves grouping queries in the self-attention mechanism based on their similarity. You would need to modify the attention mechanism in the transformer layers. Here's a simplified example:

```
from transformers import GPT2Model, GPT2Config
import torch
import torch.nn.functional as F

class GPT2WithGroupQueryAttention(GPT2Model):
    def __init__(self, config):
        super().__init__(config)
        # Add group query attention mechanism here

    def forward(self, input_ids=None, attention_mask=None, **kwargs):
        # Modify the forward pass to incorporate group query attention
        # ...
        return super().forward(input_ids, attention_mask=attention_mask, **kwargs)
```

3. Sliding Window Attention:

Sliding Window Attention is a mechanism introduced in Longformer. It allows for capturing longer-range dependencies with reduced computational cost compared to full attention. Here's a simplified example:

```
from transformers import GPT2Model, GPT2Config
import torch
import torch.nn.functional as F

class GPT2WithSlidingWindowAttention(GPT2Model):
    def __init__(self, config):
        super().__init__(config)
        # Add sliding window attention mechanism here

    def forward(self, input_ids=None, attention_mask=None, **kwargs):
        # Modify the forward pass to incorporate sliding window attention
        # ...
        return super().forward(input_ids, attention_mask=attention_mask, **kwargs)
```

TASK3

```
import torch
import torch.nn as nn
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel
from fsdp import FullyShardedDataParallel

# Define a simple model (replace with your actual model)
class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.fc = nn.Linear(10, 1)

    def forward(self, x):
        return self.fc(x)

# Replace this with your actual dataset and dataloader
# For simplicity, using a random tensor as a placeholder
train_dataset = torch.randn(100, 10)
train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=16)

def train_step(model, optimizer, criterion, data):
```

```
    optimizer.zero_grad()
    output = model(data)
    loss = criterion(output, torch.randn_like(output))  # Replace with your actual loss function
    loss.backward()
    optimizer.step()
    return loss.item()

def main():
    # Set device and initialize model
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = MyModel().to(device)

    # Replace this with your actual optimizer and criterion
    optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
    criterion = nn.MSELoss()

    # Dummy distributed training setup
    world_size = torch.cuda.device_count()
    rank = 0
    if torch.cuda.is_available():
        dist.init_process_group("nccl", rank=rank, world_size=world_size)

    # Wrap model with DDP
    if world_size > 1:
        model = DistributedDataParallel(model, device_ids=[rank])

    # Wrap model with FSDP
    # Uncomment the following lines if fsdp package is installed
    # from fsdp import FullyShardedDataParallel
    # model = FullyShardedDataParallel(model)

    # Training loop
    for epoch in range(epochs):
        model.train()
        for data in train_dataloader:
            data = data.to(device)

            # Forward and backward pass
            loss = train_step(model, optimizer, criterion, data)

        if rank == 0:
            print(f"Epoch {epoch + 1}, Loss: {loss}")

    # Cleanup
```

```python
    if torch.cuda.is_available():
        dist.destroy_process_group()

if __name__ == "__main__":
    main()
```