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

if torch.cuda.is\_available():

dist.destroy\_process\_group()

if \_\_name\_\_ == "\_\_main\_\_":

main()