Coding Assignment: Implementation and Optimization of GPT-2 Model

Task 1: - **GPT-2 Model & Checkpoints**

1. **Configuration Setup:** I defined a **GPT2Config** class to hold the configuration parameters for the GPT-2 model, including vocabulary size, maximum position embeddings, number of layers, heads, embedding dimensions, and other relevant settings.
2. **Scaled Dot-Product Attention Function:** I implemented the scaled dot-product attention function required for the multi-head attention mechanism in the Transformer. This function computes attention scores between query, key, and value tensors.
3. **Attention Components:** I created an **AttentionHead** class responsible for a single attention head and a **MultiHeadAttention** class that handles multiple attention heads by utilizing the previously defined attention head.
4. **Pointwise Feed-Forward Layer:** I constructed a **PointwiseFeedForward** class, representing the feed-forward network within the Transformer block.
5. **Transformer Block:** I defined a **TransformerBlock** class that encapsulates the operations within a single Transformer block, including multi-head attention and feed-forward layers, along with layer normalization.
6. **GPT-2 Model Architecture:** I implemented the **GPT2** class, which integrates the components defined earlier. It includes token and positional embeddings, multiple transformer blocks, and layer normalization to create the GPT-2 model.
7. **Example Usage:** In the main block, I instantiated the **GPT2Config**, created an instance of the **GPT2** model based on this configuration, generated random input tokens (**input_ids**), and obtained model predictions by passing this input through the GPT-2 model (**output**). Finally, I printed the model's output for demonstration purposes.

Task 2: - **Transformer Architectural Changes**

Rotary Positional Embedding

- **Model Size and Capabilities:** The rotary positional embeddings could potentially enhance the model's ability to capture positional information. However, the exact impact on model size depends on the implementation and size of the positional embeddings used.
- **Potential Pitfalls:** The introduction of rotary positional embeddings might increase computational complexity, potentially affecting training time and memory requirements.
- **Improvement:** The use of rotary positional embeddings might help the model learn more sophisticated positional relationships, possibly enhancing its performance in tasks reliant on sequential information.

Group Query Attention

- **Model Size and Capabilities:** Implementing group query attention could potentially facilitate capturing diverse relationships within the data by allowing different parts of the input to attend to different aspects simultaneously. The effect on model size depends on the number of groups used.
- **Potential Pitfalls:** Increasing the number of query groups might escalate computational complexity, affecting training and inference times.
- **Improvement:** Group query attention might improve the model's ability to attend to various parts of the input differently, potentially enhancing its representation capabilities.

Sliding Window Attention:

- **Model Size and Capabilities:** Sliding window attention might enable the model to focus on local contexts efficiently, aiding in tasks requiring long-range dependencies without the need for extensive computational resources.
- **Potential Pitfalls:** Increasing the window size could escalate computational complexity, especially for longer sequences. Careful tuning of the window size is necessary.
- **Improvement:** Sliding window attention might enhance the model's ability to attend to relevant information within a specific context window, potentially improving its performance in tasks requiring context-based understanding.

Task 3: - **Training Loop Implementation**

Single GPU Training Loop:

- **The code defines a basic training loop that operates on a single GPU. It checks for GPU availability and moves the model to the available GPU.**
- **It iterates through the dataset batches, performs forward and backward passes through the model, and updates the model's weights via optimization.**

Distributed Data Parallel (DDP)

- **This code snippet sets up Distributed Data Parallel (DDP) training. It initializes the process group, moves the model to the GPU, and wraps the model with DistributedDataParallel.**
- **It also adjusts the dataloader to use DistributedSampler for distributed training, ensuring each process gets unique dataset samples.**
- **The training loop structure remains similar to the single GPU setup but requires setting the epoch for the sampler at the start of each epoch to ensure different process groups receive different data samples.**

Fully Sharded Data Parallel

- **The code demonstrates a basic training loop using Fully Sharded Data Parallel (FSDP) with a simple linear model, cross-entropy loss, and SGD optimizer.**
- **It moves the model to the GPU and iterates through the data loader while performing forward and backward passes similar to single GPU training.**