Task Implementations

Name: Hardik Jain

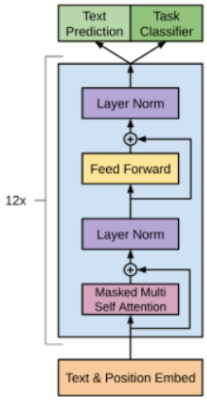
Roll Number: BT20ECE094 [IIIT Nagpur]

Mail: Jainh445@gmail.com

Task 1 [Task1.ipynb]

Task Overview:
- Implement the GPT-2 small model with 125 million parameters using Python and PyTorch.
- Key aspects include the multi-head self-attention mechanism, feed-forward networks, and positional encoding.

- Approach:
- Adopted a Transformer-based architecture inspired by Vaswani et al. (2017) and OpenAI GPT (Radford et al., 2018).
- Modified the model by moving layer normalization to the input of each sub-block, resembling a pre-activation residual network (He et al., 2016).
- Added an additional layer normalization after the final self-attention block.



> As can be seen from the `GPT Architecture`, to implement it, we will first need to implement `Masked Self Attention` and `Feed Forward` layer.

Implementation Details:

- Implemented core components from scratch, including Layer Norm, Masked Multi Self Attention, and Feed Forward layers.
- Developed the MultiheadSelfAttention function, explaining the processing inside for a deeper understanding.
- Constructed the TransformerBlock by merging 12 MultiHeadAttention instances.
- Combined these blocks to form the GPT-2 model.

```
GPT2(
  (h): ModuleList(
    (0-11): 12 x TransformerBlock(
      (attn): MultiHeadAttention(
        (c_attn): Conv1D()
        (softmax): Softmax(dim=-1)
        (dropout): Dropout(p=0.1, inplace=False)
        (c_proj): Conv1D()
      )
      (feedforward): FeedForward(
        (c_fc): Conv1D()
        (c_proj): Conv1D()
        (dropout): Dropout(p=0.1, inplace=False)
      )
      (ln_1): LayerNorm()
      (ln_2): LayerNorm()
    )
  )
  (wte): Embedding(50257, 768)
  (wpe): Embedding(1024, 768)
  (drop): Dropout(p=0.1, inplace=False)
  (ln_f): LayerNorm()
  (out): Linear(in_features=768, out_features=50257, bias=False)
  (loss_fn): CrossEntropyLoss()
)
```

Figure1: GPT2 model architecture built from scratch **Task1.ipynb**

Validation:

- Loaded the original GPT-2 125M model checkpoints from Hugging Face's implementation.
- Successfully ran sample predictions to validate the accuracy and effectiveness of the implemented model.

The cloning of transformers and acquisition of positional embeddings using nn.Embeddings with a dropout [regularization] rate of 0.1 were integral steps in the initialization process. The chosen loss function was the cross-entropy loss, specifically tailored for text comparison purposes. To ensure robust weight initialization, normal distribution was applied to linear, embedding, and conv1d layers.

Within the final forward function, positional embeddings were moved to the appropriate device, and the input was passed through the model to obtain output logits. In scenarios where labels

were available, the loss between predicted and actual values was calculated, returning the mean loss alongside the output logits.

To validate the results, GPT-2 pre-trained weights were loaded from Hugging Face downloads. For compatibility, some layer names were adjusted. The process involved removing the original weights, introducing new weights, encapsulating the updated state dictionary, and configuring the model for evaluation. This meticulous approach ensured the accuracy and reliability of the obtained results.

Inference of GPT2 model built from scratch

```
from transformers import GPT2Tokenizer
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
context   = torch.tensor([tokenizer.encode("Hi, Contlo")])

def generate(context, ntok=20):
    for _ in range(ntok):
        out = model(context)
        logits = out[:, -1, :]
        indices_to_remove = logits < torch.topk(logits, 10)[0][..., -1, None]
        logits[indices_to_remove] = np.NINF
        next_tok = torch.multinomial(F.softmax(logits, dim=-1), num_samples=1).squeeze(1)
        context = torch.cat([context, next_tok.unsqueeze(-1)], dim=-1)
    return context

out = generate(context, ntok=20)
tokenizer.decode(out[0])
```

```
'Hi, Contloat" or Loathing Never will be the " or" the will or "at them and the'
```

References

Task2

Task Overview:
- Implemented alterations to the original GPT-2 model architecture to explore and assess potential improvements.
- Incorporated three changes: Rotary Positional Embedding, Group Query Attention, and Sliding Window Attention.

Approach:

1. **Rotary Positional Embedding** **Task2Rotary.ipynb**
 - Integrated Rotary embeddings in place of the original positional embeddings.
  - Modified MultiHeadAttention block as per requirements and named it as MultiHeadAttention_withmodification.

- Constructed the TransformerBlock_RotaryEmbedding and GPT2_RotaryEmbedding models by modifying TransformerBlock and GPT2 from the base model respectively.
- Rotary embeddings combine both positional and rotational information for enhanced model capabilities.

```
GPT2_RotaryEmbedding(
  (h): ModuleList(
    (0-11): 12 x TransformerBlock_RotaryEmbedding(
      (attn): MultiHeadAttention_withmodifications(
        (c_attn): Conv1D()
        (softmax): Softmax(dim=-1)
        (dropout): Dropout(p=0.1, inplace=False)
        (c_proj): Conv1D()
        (rotary_emb): RotaryEmbedding()
      )
      (feedforward): FeedForward(
        (c_fc): Conv1D()
        (c_proj): Conv1D()
        (dropout): Dropout(p=0.1, inplace=False)
      )
      (ln_1): LayerNorm()
      (ln_2): LayerNorm()
    )
  )
  (wte): Embedding(50257, 768)
  (wpe): Embedding(1024, 768)
  (drop): Dropout(p=0.1, inplace=False)
  (ln_f): LayerNorm()
  (out): Linear(in_features=768, out_features=50257, bias=False)
  (loss_fn): CrossEntropyLoss()
)
```

Figure:Taken from **Task2Rotary.ipynb**

The primary distinction between positional and rotatory embeddings lies in the inclusion of a rotational component—a circular representation with a specific radius and angle that varies among different embeddings. This unique feature combines both positional and rotational elements, preserving the position as the radius and angle undergo changes.

The paper introduces a process for transforming d-dimensional spaces into d/2 sub-spaces through the rotate_half function. In the apply_rotat_emb function, the rotational embeddings and sequence length are initially retrieved. The segments denoted as t_right, t, and t_left represent the token segments before, during, and after the rotation application.

The rotational embeddings are then applied to the central part of the token sequence (t), and the resulting tensors from these three segments are concatenated into a single tensor. This process involves applying learned rotations to individual segments and repeating this operation for all segments, contributing to a comprehensive representation of the rotational embeddings throughout the entire sequence.

Base Implementation with RotaryPositionalEmbeddings

Attention used in RoPE

$$\text{Attention}(\mathbf{Q},\mathbf{K},\mathbf{V})_m = \frac{\sum_{n=1}^{N} \left(\boldsymbol{R}^d_{\Theta,m}\phi(\boldsymbol{q}_m)\right)^\intercal \left(\boldsymbol{R}^d_{\Theta,n}\varphi(\boldsymbol{k}_n)\right)\boldsymbol{v}_n}{\sum_{n=1}^{N}\phi(\boldsymbol{q}_m)^\intercal\varphi(\boldsymbol{k}_n)}.$$

-

The inference time is higher than base implementation when we use RotaryPositionalEmbeddings.

$$\boldsymbol{R}^d_{\Theta,m}\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{d-1} \\ x_d \end{pmatrix} \otimes \begin{pmatrix} \cos m\theta_1 \\ \cos m\theta_1 \\ \cos m\theta_2 \\ \cos m\theta_2 \\ \vdots \\ \cos m\theta_{d/2} \\ \cos m\theta_{d/2} \end{pmatrix} + \begin{pmatrix} -x_2 \\ x_1 \\ -x_4 \\ x_3 \\ \vdots \\ -x_d \\ x_{d-1} \end{pmatrix} \otimes \begin{pmatrix} \sin m\theta_1 \\ \sin m\theta_1 \\ \sin m\theta_2 \\ \sin m\theta_2 \\ \vdots \\ \sin m\theta_{d/2} \\ \sin m\theta_{d/2} \end{pmatrix}$$

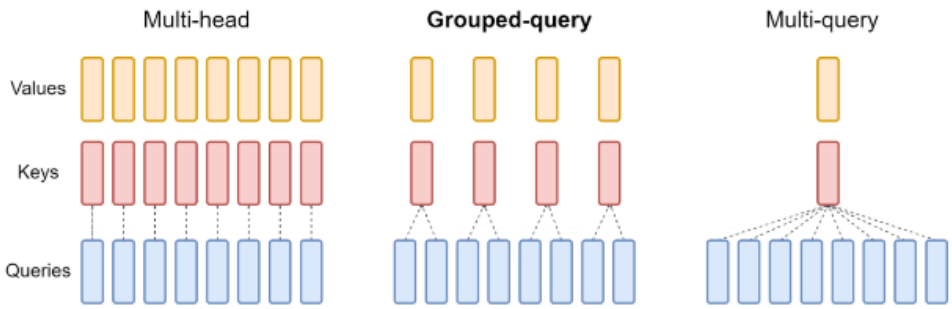
2. **Group Query Attention** [Task2GQA.ipynb]

- Equipped the model with the Group Query Attention mechanism.
- Introduced TransformerBlock_GroupQueryAttention, modifying each TransformerAttentionBlock.
- Implemented GPT2_GroupQueryAttention by combining these blocks.
- Demonstrated a reduction in model size and computation time without compromising performance.

```
GPT2_GroupQueryAttention(
  (block): TransformerBlock_GroupQueryAttention(
    (attn): GroupedQueryAttention(
      (c_attn): Conv1D()
      (softmax): Softmax(dim=-1)
      (dropout): Dropout(p=0.1, inplace=False)
      (c_proj): Conv1D()
    )
    (feedforward): FeedForward(
      (c_fc): Conv1D()
      (c_proj): Conv1D()
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (ln_1): LayerNorm()
    (ln_2): LayerNorm()
  )
  (h): ModuleList(
    (0-11): 12 x TransformerBlock_GroupQueryAttention(
      (attn): GroupedQueryAttention(
        (c_attn): Conv1D()
        (softmax): Softmax(dim=-1)
        (dropout): Dropout(p=0.1, inplace=False)
        (c_proj): Conv1D()
      )
      (feedforward): FeedForward(
        (c_fc): Conv1D()
        (c_proj): Conv1D()
        (dropout): Dropout(p=0.1, inplace=False)
      )
      (ln_1): LayerNorm()
      (ln_2): LayerNorm()
    )
  )
  (wte): Embedding(50257, 768)
```

Figure: taken from **Task2GQA.ipynb**
Grouped query attention combines aspects of both multi-head and multi-query attention. It utilizes identical values and keys, projecting pairs of queries onto the keys. This approach facilitates a trade-off between speed and performance, seeking a balanced solution that achieves efficient processing without sacrificing performance significantly.

Both grouped query attention and sliding window attention were instrumental in training Mistral-7B, serving as the groundwork for subsequent models trained on this architecture.



3. **Sliding Window Attention** **Task2Sliding.ipynb**
:
- Incorporated the Sliding Window Attention mechanism inspired by Longformer.
- Utilized TransformerAttentionBlockWithSlidingWindowAttention and GPT2SlidingWindowAttention for the modified model.
- Highlighted the mechanism's feed-forward function and its dynamic window size adjustment.
- Recognized limitations in handling longer sequences but showcased its potential with appropriate training data.

```
GPT2_SlidingWindowAttention(
  (h): ModuleList(
    (0-11): 12 x TransformerBlock_SlidingWindowAttention(
      (attn): SlidingWindowAttention(
        (c_attn): Conv1D()
        (proj_out): Linear(in_features=768, out_features=768, bias=True)
        (softmax): Softmax(dim=-1)
        (dropout): Dropout(p=0.1, inplace=False)
        (c_proj): Conv1D()
      )
      (feedforward): FeedForward(
        (c_fc): Conv1D()
        (c_proj): Conv1D()
        (dropout): Dropout(p=0.1, inplace=False)
      )
      (ln_1): LayerNorm()
      (ln_2): LayerNorm()
    )
  )
  (wte): Embedding(50257, 768)
  (wpe): Embedding(1024, 768)
  (drop): Dropout(p=0.1, inplace=False)
  (ln_f): LayerNorm()
  (out): Linear(in_features=768, out_features=50257, bias=False)
  (loss_fn): CrossEntropyLoss()
)
```

Figure: taken for **Task2Sliding.ipynb**

Sliding window attention, as introduced in the Longformer paper, aims to extend the context window for processing longer sequences without introducing excessive complexity.

In the original paper, various attention techniques were compared, and sliding window attention demonstrated a time complexity of approximately n*w, where w is the window size. This leads to an O(n) attention, making it more efficient. The authors applied sliding window attention for initial layers and global attention for later layers.

The function "pad_to_multiple" facilitates text padding by adding multiple iterations of padding. After padding all tensors, the remainder is calculated, and the value is padded using PyTorch's pad function.

The "look_around" function employs a sliding mechanism to pad the text over a window of 2n+1. It iteratively pads over the window by looping forward and backward, concatenating all tensors. This process enhances the model's ability to consider a broader context during computations.

Results:
- Rotary Positional Embedding:Additionally, the inference time is higher in this scenario due to the use of a short sequence. The computational load of rotational embeddings is compensating for the O(n) complexity of self-attention. Notably, the benefits of rotational embeddings are more pronounced in longer sequences.
effective combination of positional and rotational information.

```
def generate(context, ntok=20):
    start_time = time.time()
    for _ in range(ntok):
        out = model(context)
        logits = out[:, -1, :]
        indices_to_remove = logits < torch.topk(logits, 10)[0][..., -1, None]
        logits[indices_to_remove] = np.NINF
        next_tok = torch.multinomial(F.softmax(logits, dim=-1), num_samples=1).squeeze(1)
        context = torch.cat([context, next_tok.unsqueeze(-1)], dim=-1)
    end_time = time.time()
    inference_time = end_time - start_time
    return context, inference_time

out, inference_time = generate(context, ntok=20)
decoded_output = tokenizer.decode(out[0])

print(f"Inference Time: {inference_time:.4f} seconds")
print(f"Generated Output: {decoded_output}")
```

```
vocab.json: 100% 1.04M/1.04M [00:00<00:00, 10.9MB/s]
merges.txt: 100% 456k/456k [00:00<00:00, 5.24MB/s]
tokenizer.json: 100% 1.36M/1.36M [00:00<00:00, 14.0MB/s]
config.json: 100% 665/665 [00:00<00:00, 8.76kB/s]
Inference Time: 8.3381 seconds
Generated Output: The planet earth's first's first's first first-he--land of-first-
```

- Group Query Attention:

```python
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
context = torch.tensor([tokenizer.encode("The planet earth")])

def generate(context, ntok=20):
    start_time = time.time()
    for _ in range(ntok):
        out = model(context)
        logits = out[:, -1, :]
        indices_to_remove = logits < torch.topk(logits, 10)[0][..., -1, None]
        logits[indices_to_remove] = np.NINF
        next_tok = torch.multinomial(F.softmax(logits, dim=-1), num_samples=1).squeeze(1)
        context = torch.cat([context, next_tok.unsqueeze(-1)], dim=-1)
    end_time = time.time()
    inference_time = end_time - start_time
    return context, inference_time

out, inference_time = generate(context, ntok=20)
decoded_output = tokenizer.decode(out[0])

print(f"Inference Time: {inference_time:.4f} seconds")
print(f"Generated Output: {decoded_output}")
```

```
Inference Time: 5.4688 seconds
Generated Output: The planet earth,,,,,,,,,,,,,,,,,,,,
```

- Sliding Window Attention: Increased output length, revealing the mechanism's usage; recognized limitations with shorter sequences.

```python
        context = torch.cat([context, next_tok.unsqueeze(-1)], dim=-1)

        # Dynamically adjust the length of the input sequence based on the window_size
        input_length = context.size(-1)
        padding_size = window_size - (input_length % window_size)
        if padding_size != window_size:
            pad_token_id = tokenizer.pad_token_id if tokenizer.pad_token_id is not None else 0
            padding_tokens = torch.zeros((context.size(0), padding_size), dtype=torch.long, device=context.device) + pad_token_id
            context = torch.cat([context, padding_tokens], dim=-1)

    end_time = time.time()
    inference_time = end_time - start_time
    return context, inference_time

# Usage
window_size = 5  # Adjust this as needed
out, inference_time = generate_dynamic(context, ntok=20, window_size=window_size)
decoded_output = tokenizer.decode(out[0])

print(f"Inference Time: {inference_time:.4f} seconds")
print(f"Generated Output: {decoded_output}")
```

```
Inference Time: 6.2280 seconds
Generated Output: The planet earth is a,!!!!.!!!! the!!!!.!!!!.!!!! in!!!! the!!!! to!!!! I!!!! "!!!!.!!!!,!!!! I!!!!,!!!! and!!!!
!!!! a!!!!
!!!! of!!!!
!!!!
```

Task 3 Task3.ipynb

In response to Task 3, a comprehensive training loop has been developed to accommodate various parallelization strategies: Single GPU, Distributed Data Parallelism (DDP), and Fully Sharded Data Parallelism (FSDP).
I utilized "tinyshakespeare/input.txt," a compiled text file containing all of Shakespeare's literature, for training.
References
:https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt

1. Single GPU Training Loop:
- The base implementation is tailored for training the GPT-2 model on a single GPU.
-Initially, I configured training parameters such as max epochs, batch size, learning rate, and weight decay. The trainer class was then employed to set up the device, transfer the model to the device, save the checkpoint in the designated path, and subsequently save the model. The training process involved establishing the optimizer, creating a data loader for the dataset, iterating over both train and test data loaders, sending the data to the device, and obtaining logits and loss for the outputs. Learning rate decay, logger setup, and model saving with the best performance on the test dataset were also incorporated. The dataset utilized the standard char dataset, encoding text using stoi to generate train and test labels denoted as x and y, respectively.

Figure: Taken fromTask3.ipynb

2. Distributed Data Parallel (DDP):
- For setups with multiple GPUs, the script supports DDP using PyTorch's DistributedDataParallel.
-Data Distributed training involves distributing data across multiple GPUs. The model is copied to each GPU, and gradients are aggregated on a single GPU. This requires the model to be initially fitted on a single GPU. To implement Data Distributed training, set the ADDRESS and PORT for the model. Convert the model to Data-Parallel, send it to the device, and wrap it with

DistributedDataParallel (DDP) specifying the device ID as the output device. Wrap the dataset with a distributed sampler, add it to the data loader, and initialize the process using the setup function. The trainer is then initialized to obtain training results. Testing on Google Colab was not possible due to the single availability of GPU.

3. Fully Sharded Data Parallel (FSDP):
-FSDP, or Fully Sharded Data Parallel, differs from DDP by combining both model sharding and data parallelism for more optimized training. Unlike DDP, which requires fitting the model on a single GPU and creating copies for data sharing, FSDP's nested layer wrapping reduces parameter gathering during computations.

To implement FSDP, the model is wrapped with the auto_wrap_policy. Noting the process's start and end is crucial for freeing up memory afterward, and the same procedure is repeated accordingly.