Building GPT-2 Transformer-Based Model from Scratch

- Team Members:

1. **Basmala Ahmed** **2205192**
2. **Lujan Hussam** **2205121**
3. **Shrouk Badr** **2205132**
4. **Ahmed Muhammed Saber** **2205098**
5. **Abdelrahmn Muhammed Abdelkader** **2205237**

- This project implements a simplified version of the GPT-2 architecture from scratch using PyTorch. The main goal is to understand the internal workings of transformer-based language models and to manually construct all critical components including multi-head attention, positional encoding, feed-forward layers, and residual connections.

Unlike pre-built libraries like Hugging Face Transformers, this model avoids any use of high-level transformer APIs and builds everything manually to offer deeper learning insights.

- Model Architecture

The following components were implemented from scratch to mimic GPT-2 (Small):

1. **Positional Encoding**
   Positional information was injected into token embeddings using sinusoidal functions (`sin`, `cos`) as proposed in the original Transformer paper. This allows the model to learn word order without recurrence.

2. **Multi-Head Self-Attention**
   We implemented scaled dot-product attention across multiple heads. Queries, keys, and values are linearly projected, and attention scores are computed with proper scaling. The outputs are then concatenated and passed through a final linear layer.

3. **Feed-Forward Neural Network**
   Each transformer block contains a feed-forward sublayer applied position-wise using two linear layers and a GELU activation.

4. **Layer Normalization**
   Layer normalization is applied before both the attention and feed-forward sublayers (as in GPT-2's pre-norm design).

5. **Residual Connections**
   Residual (skip) connections are included around both the attention and feed-forward components to stabilize training and improve gradient flow.

6. **Stacked Decoder Blocks (12 layers)**
   The model consists of 12 decoder blocks, each combining self-attention, feed-forward networks, layer norm, and residuals—faithfully mimicking GPT-2 small.

7. **Token & Output Embeddings**
   The model uses token embeddings (`nn.Embedding`) and projects the final outputs back to vocabulary size using a linear head for next-token prediction.

- Dataset and Preprocessing

- **Dataset**: TinyStories, a collection of simple, story-like English texts designed for training compact language models.
- **Preprocessing**: Tokenization was done using the pretrained GPT-2 tokenizer. Each sample is converted into sequences of fixed length (128 tokens).
- **Samples used**: For training, we selected a subset of `max_samples = 10000` lines from the dataset due to memory limitations.

Each line from the dataset is turned into overlapping `(input, target)` pairs for next-token prediction.

- Training Setup

- **Loss Function**: CrossEntropyLoss
- **Optimizer**: AdamW
- **Batch Size**: 8
- **Epochs**: 5
- **Device**: GPU (when available)

The training loop runs for 5 full passes (epochs) over the dataset, calculating average loss per epoch. The training is conducted using standard PyTorch `DataLoader` and `tqdm` progress bars for visualization.

- Evaluation – Perplexity

To evaluate the model's confidence in predicting the next token, we calculated **Perplexity**:

e ^ Average loss = Perplexity

- Final Perplexity: ~1.23

This is an excellent result, showing the model is very confident in its predictions—especially considering the small sample size and limited training time.

-Text Generation

After training, the model was used to generate text based on the prompt:

```
"Once upon a time"
```

We sampled 5 outputs using **multinomial sampling** (not greedy decoding) to enhance creativity and diversity.

Example Output (trimmed):

```
Sample 1:
Once upon a timeloo. Sometimes things die and was full of fruits and up at the thanked grassy was full of water bubOnt Grandma lovesOldInsp set coils total on top. Daisy sighed and tol

Sample 2:
Once upon a time Conj. McCluggled under at home from then said, "Can you forgive me byimony smiled and told my friends again?" Her friends again?" Her friends looked at her. They said

Sample 3:
Once upon a time garden something special momentAngNAT 1893 MET went to get closer. But we can't want to me?" The furry animal looked up at her and started to move. Sarah was by her ar

Sample 4:
Once upon a time As and pushed and very serious.video very healthy because she ate lots of fruits and people in each other. She stopped and Lat itself A, furry animal. Sarah was very e

Sample 5:
Once upon a time stepped forward play together and stepped along. suspense Sarah decided to soak in theCapture books would be my friends again?" Her friends again?" Her friends looked
```

- Observations and Challenges

- Training on **100% of the dataset** is not feasible on free Colab due to memory and session time constraints.
- Increasing `max_samples` to 250k or more significantly improves generation quality.
- Using sin/cos positional encoding instead of learned embeddings matches the project requirements better and teaches more about the theory.

- Conclusion

This project successfully builds a GPT-2-style model from scratch with:

- 12 decoder layers
- 12 attention heads
- Manual implementation of all transformer blocks
- Perplexity evaluation
- Text generation

All components were written manually using low-level PyTorch, without relying on any prebuilt transformer modules. The results demonstrate that even with limited data, a properly built GPT-2 architecture can learn meaningful patterns and generate coherent text.

- Detailed Explanation of the GPT-2 From-Scratch Implementation

Step 1: Import Libraries

- Import PyTorch modules for building neural networks and optimization.
- Import math for mathematical functions.
- Import tokenizer from HuggingFace Transformers for tokenizing text data.
- Use tqdm for progress visualization during training.

Step 2: Define Positional Encoding

- Create fixed sinusoidal positional embeddings to add token position info to word embeddings.
- This helps the model understand the order of tokens in the sequence since Transformers do not process sequences sequentially.

Step 3: Implement Multi-Head Self-Attention

- Define linear layers to project inputs into Query, Key, and Value matrices.
- Split these into multiple heads for capturing diverse relationships.
- Calculate scaled dot-product attention scores between queries and keys.
- Apply softmax to get attention weights, then combine with values to get context-aware token representations.
- Concatenate heads and apply a final linear layer.

Step 4: Define Feed-Forward Network

- Add two fully connected layers with GELU activation between them.
- Applies non-linear transformation independently to each token embedding, enhancing model capacity.

Step 5: Build Transformer Block

- Combine Multi-Head Self-Attention and Feed-Forward submodules with Layer Normalization and residual connections.
- These blocks form the core building unit of the Transformer, stacked multiple times for depth.

Step 6: Construct the GPT-2 Model

- Embed input tokens using a learned token embedding layer.
- Add positional encodings to token embeddings.
- Pass through a sequence of Transformer blocks (typically 12).
- Normalize output and project to vocabulary size to predict next tokens.

Step 7: Prepare the Dataset

- Load raw text lines from file.
- Tokenize each line using pretrained GPT-2 tokenizer.
- Extract fixed-length token sequences as inputs, and corresponding shifted sequences as targets (next-token prediction).
- Store sequences as tensors for DataLoader batching.

Step 8: Initialize Training Setup

- Move model to available device (GPU if possible).
- Define optimizer (AdamW) and loss function (Cross-Entropy).
- Create DataLoader for batch training.

Step 9: Train the Model

- Iterate over multiple epochs.
- For each batch:
    - Forward pass inputs through model.
    - Compute loss against targets.
    - Backpropagate gradients.
    - Update model parameters.
- Track and print average loss per epoch.

Step 10: Compute Perplexity

- Set model to evaluation mode.
- Compute average cross-entropy loss over dataset without updating weights.
- Calculate perplexity as exponential of average loss, measuring model’s predictive uncertainty.

Step 11: Generate Text Samples

- Given a prompt, tokenize it and feed to model.
- Iteratively sample next tokens from the model’s predicted probability distribution.
- Append generated tokens to the input and repeat until max tokens are produced.
- Decode tokens back to text, producing coherent generated samples.

- Key Components of the Model

Positional Encoding

Since the Transformer architecture processes input tokens in parallel without inherent knowledge of their order, the model adds **positional encodings** to the token embeddings. These encodings use sine and cosine functions with varying frequencies to uniquely represent each position in the input sequence. Adding these to token embeddings allows the model to understand token order and sequence structure.

Multi-Head Self-Attention

Self-attention allows the model to focus on different parts of the input sequence when processing each token. The multi-head attention splits the embedding into multiple smaller parts ("heads"), computes attention separately for each, and then concatenates the results. This enables the model to capture diverse relationships and dependencies within the input at different representation subspaces simultaneously.

The attention weights are computed using a scaled dot product between queries and keys, normalized with softmax, then applied to the values. This allows the model to weigh the importance of different tokens dynamically.

Note: In a GPT-2 model, causal masking is usually applied to prevent tokens from attending to future tokens (to keep predictions autoregressive), but this code does not explicitly show that part.

Feed Forward Network (FFN)

Each Transformer block contains a position-wise feed-forward network which consists of two linear transformations separated by a non-linear activation function (GELU). This layer increases the model's representational power by applying complex non-linear transformations to each token's embedding independently.

Transformer Block

The Transformer block combines multi-head self-attention and the feed-forward network with residual connections and layer normalization. The residual connections help with gradient flow during training, making it easier to train very deep models. Layer normalization stabilizes training and improves convergence.

The architecture follows a **pre-normalization** style where layer normalization is applied before the attention and feed-forward layers.

GPT-2 Model Structure

The full GPT-2 model stacks multiple Transformer blocks sequentially (commonly 12 blocks). It first converts input token indices into embeddings, adds positional encodings, then passes the result through the stacked Transformer blocks. A final layer normalization is applied, followed by a linear layer that projects the hidden states back to the vocabulary size to produce logits (raw prediction scores) for each token.

During training, these logits are used to compute the loss against the true next tokens.

3. Data Preparation and Loading

The dataset class reads a text file containing sentences or documents. It tokenizes each line using a pre-trained tokenizer to convert words into token IDs.

From the tokenized text, the dataset generates many overlapping training samples. Each sample consists of an input sequence and its corresponding target sequence, which is the input sequence shifted by one token forward. This prepares the model to learn next-token prediction.

Samples are converted into tensors and stored for batching during training.

4. Training Loop (Summary)

- The model is trained in batches over many epochs.
- Each batch consists of multiple token sequences.
- For each batch, the model predicts the next token logits for every position.
- A loss function (usually cross-entropy) compares the predicted logits with the true next tokens.
- Gradients are computed by backpropagation.
- The model's parameters are updated using an optimizer (e.g., Adam).
- Progress bars (e.g., via tqdm) show training progress for monitoring.

- Important Notes

- The model size, number of heads, hidden sizes, and number of layers can be adjusted depending on computational resources.
- Positional encoding is fixed and non-learnable in this implementation.
- The model does not explicitly show causal masking; adding it is crucial for proper autoregressive behavior during training and generation.
- Using a pre-trained tokenizer allows consistent and efficient tokenization aligned with the model's vocabulary.
- This code provides a foundation for GPT-2 style language modeling but can be extended with additional features like learning rate scheduling, checkpoint saving/loading, and generation functions.