Implementation of a Language Model

K Vivek Kumar CS21BTECH11026
Jarupula Sai Kumar CS21BTECH11023
Akkasani Yagnesh Reddy CS21BTECH11003
Nimmala Avinash CS21BTECH11039
Saurabh Dhavjekar EE24RESCH01003

CONTENTS

***Abstract*—Language models have gained significant popularity in recent years, revolutionizing fields such as creative text generation, summarization, and reasoning. Tools like ChatGPT have become essential for users seeking precise answers, offering a more direct and efficient alternative to traditional search engines. In this work, we explore the fundamental architecture behind such models — the transformer blocks and attention mechanisms [1]. Despite training on limited computational resources, we successfully built and trained a functional language model from scratch. Even with minimal data, measured in kilobytes, our model was able to generate contextually relevant responses to various prompts. With access to larger datasets and more computational power, this model could excel in tasks like text completion and summarization, showcasing the immense potential of language models when scaled.**

I. INTRODUCTION

Inspired by the insights provided by recent work on large language models [2], we explore the inner workings of such models, particularly focusing on their ability to translate tasks into English and perform reasoning. These models, trained on vast corpora with a focus on major languages like English, have demonstrated impressive capabilities in a wide range of tasks. Motivated by this, we set out to design and implement a decoder-based transformer model [3] to gain a deeper understanding of how language models operate.

Our approach involved carefully constructing the various phases of the model architecture, integrating them with the correct dimensionalities to ensure proper functionality. Through this implementation, we made several observations about the model's performance, its responses to different inputs, and the training procedures. These insights provide valuable contributions to the ongoing exploration of transformer-based models and their applications in context processing.

II. IMPLEMENTATION

In this section, we describe the implementation details of the various components involved in our decoder model and its training process. Each key class and its functionality are explained below.

A. Tokenizer

Initially, we developed a custom tokenizer from scratch to meet the specific needs of our research. The primary objective was to create a tokenization class based on the Byte-Pair Encoding (BPE) [4] method, which would handle sequences of tokens efficiently. This custom tokenizer integrated with the dataset pipeline, enabling it to process raw text, tokenize it into subword units, assign unique token IDs to frequent tokens, and prepare the data for model training. Additionally, we implemented utility functions for encoding and decoding tokenized text to streamline both the training and evaluation phases.

However, after evaluating the performance of our custom tokenizer, we transitioned to using the pre-trained GPT-2 tokenizer from the tiktoken library in Python. This switch was made to leverage the better performance and efficiency of the pretrained tokenizer. To integrate it into our workflow, we created a Tokenizer class that wraps around the GPT-2 tokenizer, providing essential functionality for encoding and decoding tokenized text.

B. LayerNorm and FeedForward

Layer Normalization stabilizes training by normalizing the input to each sub-layer. Implemented with LayerNorm using dimension `emb_dim`, it is applied twice in each block: once before MHA (norm1) and once before FFN (norm2), ensuring

consistent scaling and improving convergence. The Feed-Forward Network (FFN) applies two linear transformations with a non-linearity in between, enhancing token representations. It consists of two nn.Linear layers and an activation function (e.g., GeLU), introducing non-linear transformations to the data.

C. Multi-Head Attention

The positional vector embedding are given as input to the multi head attention model, we are using 12 heads in this model simultaneously, Each head acts like self attention as follows, these vectors are multiplied with Query matrix and Key matrix which outputs query vectors, key vectors for every token. Now we will compute attention scores by multiplying these vectors for every pair which builts matrices (tokens vs tokens). If a model is training on a input, it trains on the subsequence of that text as well according to Multi Head attention, these texts accelerates the training of the model, but it has a disadvantage, i.e, while training the model to predict the next word using these extra inputs, the latter ones affect the former ones because the latter token will reveal the next word which is inefficient, to avoid this we have used masking techniques for those pairs of tokens which is nothing but bottom left lower triangle of the matrix by making these values infinite(Makes the probability 0 after softmax). Then we have softmaxed the matrix to get the probabilities which are then multiplied with a value matrix which actually projects the latter token based on its probabilities with former tokens. We then sum up the values we got from all the heads followed by layer normalization.

D. Transfomer Block and Language Model Class

The Token Embedding layer converts token indices into dense vector representations using nn.Embedding with input size `vocab_size` and output size `emb_dim`. This maps discrete tokens into a continuous space, allowing the model to learn meaningful relationships between tokens.

The Positional Embedding layer adds positional information to token embeddings by using nn.Embedding with input size `context_length` and output size `emb_dim`. This ensures that the model captures the order of tokens in a sequence. The resulting positional embeddings are summed with token embeddings.

Dropout is applied to the summed embeddings using nn.Dropout with a rate of `drop_rate`. This regularization technique prevents overfitting by randomly setting some embedding values to zero during training, improving the model's ability to generalize.

Each transformer block includes Multi-Head Attention (MHA), which calculates attention weights across the sequence using MultiHeadAttention with input/output dimension `emb_dim`, `n_heads` heads, optional bias `qkv_bias`, and dropout `drop_rate`. This allows the model to focus on different parts of the sequence simultaneously, capturing contextual relationships between tokens.

Residual Connections are used to add the input of each sub-layer back to its output, implemented by summing the input and output of both MHA and FFN. This improves gradient flow and prevents the vanishing gradient problem, allowing deeper networks to train effectively.

Dropout within the transformer blocks, implemented using nn.Dropout with rate `drop_rate`, is applied after MHA and FFN to further regularize the model, reducing overfitting and improving generalization during training.

Each transformer block concludes with a Linear Head, projecting the output of the block to the vocabulary size for token prediction. Implemented as `nn.Linear(emb_dim, vocab_size, bias=False)`, it produces logits for predicting the next token in the sequence.

E. Training Process

- **Model and Optimizer Initialization:**
  - Load the model architecture defined in the `LanguageModel` class.
  - Initialize the `AdamW` optimizer with learning rate and weight decay.
- **Data Preparation:**
  - Load the input text data from the provided file path.
  - Split the data into training (90%) and validation (10%) sets.
  - Use the `create_dataloader` function to create data loaders for training and validation.
  - Tokenize the text data using the tokenizer.
- **Training Loop:**
  - For each epoch:
    - Loop through the training data in batches.
    - For each batch:
      - Compute the loss using the `compute_loss` function.
      - Backpropagate the loss and update model parameters using the optimizer.
    - Periodically evaluate the model on the training and validation datasets:
      - Compute training and validation losses using the `evaluate_model` function.
      - Log the losses and number of tokens processed.
    - Generate sample text after each epoch using the `generate_sample_text` function.
- **Loss Evaluation:**
  - Use the `evaluate_loss` function to calculate the average loss for the training and validation datasets.
  - Mask padding tokens during the loss computation using `torch.nn.functional.cross_entropy`.
- **Text Generation:**
  - Use the `generate_text` function to generate new text from a given initial prompt.
  - Decode the generated tokens to readable text using the tokenizer.

- **Training Visualization:**
  - Plot the training and validation losses across epochs using `matplotlib`.
  - Save the plot as `loss.png`.

F. Dataset

Data that is used to train our model is the story "The Selfish Giant". And also we also wrote a translator script from `deep_translator` to convert the data we had in english to other languages.We translated the story into other Languages like English, French, Spanish, Chinese, Japanese, German, Italian, Hindi. Models can be trained on each of this languages separately for making further observations.

III. Model Parameters

Through extensive experimentation with different training procedures, we arrived at the following model parameters:

- Vocabulary Size (`vocab_size`): 50,257
- Context Length (`context_length`): 256
- Embedding Dimension (`emb_dim`): 768
- Number of Attention Heads (`n_heads`): 12
- Number of Layers (`n_layers`): 12

The vocabulary size is based on the tokenization provided by the `GPT-2` tokenizer. These parameters were carefully chosen and play a critical role in the model's performance, forming the foundation for the key observations and analyses discussed in the subsequent sections.

IV. Observations

A. Observations during Training

Several observations were made while training the model with varying epoch counts, model parameters, and training settings. These observations provide insights into the model's performance and resource utilization under different conditions.

- When training on a local machine with 8GB RAM, an Intel i5 processor, and no GPU, the model was able to handle a batch size of 4 and 25 epochs without significant issues. However, the architecture performed more efficiently with a batch size of 2 and 15 epochs, showing better convergence and stability during training.
- The validation loss showed a tendency to converge, but it did not reach a significant plateau, indicating that further training or adjustments to model parameters might be required for better performance.

B. Observations during Testing

The following observations were made based on the outputs generated by the model after training with different epoch counts and parameters:

- When trained for a small number of epochs, the model's outputs were repetitive, often consisting of the same sequence of words that were irrelevant to the query context. This issue arose because the model tended to generate frequent tokens from the training data, resulting in responses with longer lengths before encountering a newline character.
- For epoch counts in the range of 10 to 12, the model began to exhibit a better understanding of the query context. However, it still did not consistently generate responses that used exact words from the context. The output was more coherent but lacked precision in using the context-specific terms.
- When the model was trained for higher epoch counts and with continued training, it demonstrated improved contextual understanding. The responses began to incorporate more contextually appropriate words, although the output was still somewhat mixed. Additionally, the response lengths before reaching the newline character were shorter compared to earlier epochs, and the model started utilizing less frequent tokens observed during training.

V. Results

The convergence of the validation loss in one of our experiments is shown below, and this pattern is similar across all other training experiments.

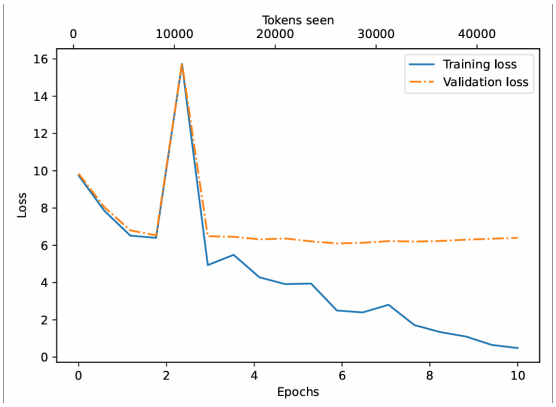

Fig. 1. Convergence of the validation loss as the number of tokens in the dataset increases.

To ensure consistent and interpretable results, we employed a specific strategy for generating model responses. For testing purposes, responses were generated only until a newline character token was encountered. The queries were designed as incomplete sentences extracted directly from the input data, terminating at a frequent noun. This approach allowed us to observe how the model incrementally completes the input. Below are examples of the observed outputs:

- **Input Query:** The children used to
  Expected Output: go and play...
- **Input Query:** The children used to go and
  Expected Output: play in the Giant's...
- **Input Query:** The children used to go and play in the
  Expected Output: Giant's garden.
- **Input Query:** The children used to go and play in the Giant's
  Expected Output: garden.

Additionally, we show the model's outputs at various epochs.

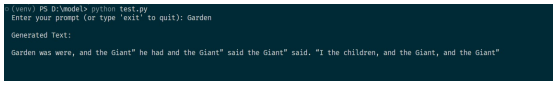

Fig. 2. Model responses after training for 3 epochs. The output contains repeated phrases and contextually irrelevant tokens.

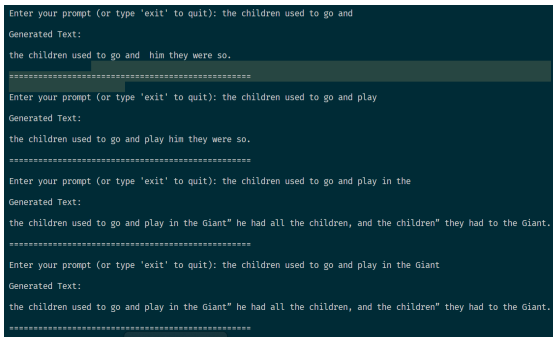

Fig. 3. Model responses after training for 5 epochs. The output size improved, but the tokens remain out of context.

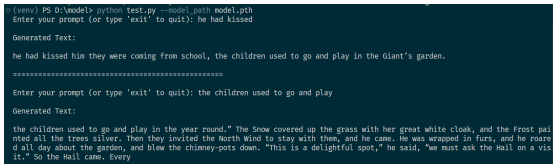

Fig. 4. Model responses after training for 12 epochs. Contextually relevant words are generated, but they are not directly aligned with the query's answer.

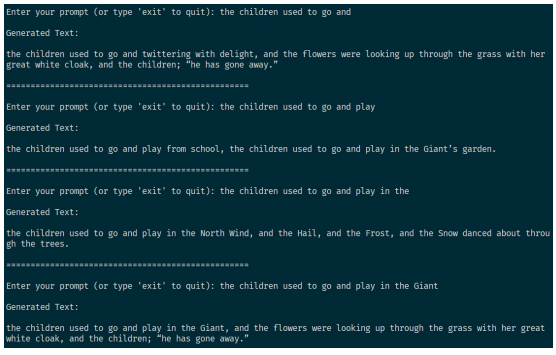

Fig. 5. Model responses after training for 15 epochs. The output improved, with more contextually relevant words generated, closely related to the query's answer.

VI. Future Work

This project opens several avenues for further enhancement and exploration. Below are some key directions for future work:

- **Dataset Expansion and Multilingual Training:** One potential extension is to train the model on various translated datasets. By examining the model's performance across different languages, we can identify which language(s) the model requires the least number of epochs or parameters to achieve optimal accuracy and context understanding.
- **Model Architecture Enhancements:** To improve the model's ability to capture cross-lingual dependencies, an additional encoder layer could be incorporated. This would allow the model to learn more sophisticated attention mechanisms, potentially improving its performance on multilingual tasks.
- **Parameter Tuning:** Further experiments with increasing model parameters could be conducted to evaluate how the model's performance scales. Observations regarding its efficiency and accuracy with different configurations would provide valuable insights into the trade-off between model size and performance.
- **Tokenizer Optimization:** Since we have developed a custom tokenizer for this model, further optimization of this tokenizer could be explored. This could include fine-tuning it to generate more contextually relevant tokens, potentially improving the model's overall performance and efficiency.
- **Logit Analysis:** As suggested in the *LogitLens* paper [2], performing a detailed logit analysis could be a valuable step. Although we currently print the logits during testing into the `logits.txt` file, further analysis using logit visualization tools could offer deeper insights into the model's decision-making process and areas for improvement.
- **Open-Source Contribution:** This repository serves as a foundation for those interested in building language models from scratch. In the future, it could be enhanced with additional documentation, tutorials, and examples to further facilitate adoption and contributions from the broader research community.

These directions represent just a few of the many potential avenues for advancing the current model. By exploring these, we can further refine the model's capabilities, performance, and versatility across a range of linguistic tasks.

VII. Contributions

- **K Vivek Kumar (CS21BTECH11026)**: Motivated and Managed the project, and worked on the Transformer block and Attention mechanisms. Training and Testing procedures handled with proper parameter testing.
- **Saurabh Dhavjekar (EE24RESCH01003)**: Designed the Tokenizer and contributed to the architectural aspects of the model.
- **Jarupula Sai Kumar (CS21BTECH11023)**: Focused on implementing Layer Normalization and Multihead Attention components.
- **Akkasani Yagnesh Reddy (CS21BTECH11003)**: Created the Transformer block diagram and contributed to the Language Model design.

- **Nimmala Avinash (CS21BTECH11039)**: Led the implementation of training procedures and contributed to Multihead Attention design.

VIII. Conclusion

This project offered invaluable insights into the intricate workings of language models, deepening our understanding of their architecture and behavior. Beyond just building a functional model, the process enriched our appreciation for the challenges and possibilities inherent in designing systems capable of generating coherent and contextually relevant text. Looking ahead, we are eager to extend this work by exploring the future directions outlined earlier, aiming to refine the model and broaden its applicability. As language models continue to revolutionize technology and everyday life, contributing to their evolution has been both a rewarding and inspiring journey.

References

[1] S. N. P. N. U. J. J. L. G. A. A. K. . P. I. Vaswani, A., "Attention is all you need," *arXiv preprint arXiv:1706.03762v*, 2024, accessed: 2024-11-29. [Online]. Available: https://arxiv.org/abs/1706.03762

[2] G. M. R. W. Chris Wendler, Veniamin Veselovsky, "Do llamas work in english? on the latent language of multilingual transformers," *arXiv preprint arXiv:2402.10588*, 2024, accessed: 2024-11-29. [Online]. Available: https://arxiv.org/abs/2402.10588

[3] N. K. S. T. . S. I. Radford, A., "Improving language understanding by generative pre-training. openai," 2018, accessed: 2024-11-29.

[4] J. L. G. L. D. T. V. M. S. R. Vilém Zouhar, Clara Meister, "Aformal perspective on byte-pair encoding," *arXiv preprint arXiv:2306.16837*, 2024, accessed: 2024-11-29. [Online]. Available: https://arxiv.org/abs/2306.16837