Implement a LLM-Lite

Jou-Chi Huang
University of Oregon

***Abstract*—This project presents a miniature large language model (LLM) inspired by ChatGPT, designed to explore transformer architectures under limited computational resources. The WikiText-2-raw-v1 dataset is preprocessed using a custom Byte Pair Encoding (BPE) tokenizer, and a transformer model is implemented from scratch with efficient data loading. After initial training with baseline settings, a hyperparameter search identified the best configuration: learning rate 0.001, six layers, four attention heads, and embedding dimension 128, achieving a perplexity of 1.17. The model is evaluated through tasks such as text generation, prompt sensitivity, and head ablation. Training dynamics are visualized using learning rate, loss, and gradient norm curves. This work offers practical insights into building and optimizing LLMs in resource-constrained environments.**

***Index Terms*—Large Language Model, Transformer, Text Generation, Tokenization, Perplexity, Resource-constrained Systems**

I. Introduction

Large Language Models (LLMs) such as GPT-4, Gemini, Claude, LLaMA, Gemma, and Grok have demonstrated exceptional performance across diverse natural language processing tasks, including grammar correction, information retrieval, mathematical reasoning, code generation, and creative writing. Their integration into everyday applications is reshaping how humans interact with technology, particularly in fields such as education, communication, and software development.

Despite their widespread impact, most state-of-the-art LLMs remain proprietary or partially closed-source, restricting access to their implementation details, training data, and optimization techniques. This lack of transparency presents a barrier for students, researchers, and developers seeking to understand, reproduce, or extend these models. Consequently, the development and control of advanced LLMs remain concentrated within a few organizations, limiting opportunities for open research and innovation.

To address this challenge, the present project implements a lightweight, fully transparent LLM from scratch using publicly available tools and datasets. The development process—covering data preprocessing, tokenizer construction, model design, training strategies, and evaluation—is documented in detail. This project aims to serve as both a learning resource and a practical guide for those interested in building transformer-based language models under resource-constrained settings, contributing to the broader goal of democratizing access to LLM technologies.

II. Background

Large Language Models (LLMs) have emerged as transformative technologies in artificial intelligence, excelling in language understanding, reasoning, and generation. Prominent examples include GPT-4, Gemini, Claude, LLaMA, Gemma, and Grok—each developed with distinct design priorities.

GPT-4 (OpenAI) is a multimodal model supporting both text and image inputs, known for its strong performance in reasoning, code generation, and domain-specific tasks. It features a context window of up to 32,000 tokens and achieves top-tier results on standardized benchmarks.

Gemini (Google DeepMind)—formerly PaLM 2—is optimized for productivity, planning, and retrieval tasks, with some variants supporting multimodal inputs and web-integrated reasoning. It is closely integrated with Google's ecosystem.

Claude (Anthropic) emphasizes safety and alignment, applying Constitutional AI to encourage helpful and responsible outputs. It performs competitively in instruction following, legal analysis, and programming tasks.

LLaMA (Meta) provides high-performance models tailored for research purposes. While not fully open-source, model weights are accessible by request and have gained significant adoption in academic and open communities.

Gemma (Google) is a lightweight, open-weight model designed for local deployment in limited-resource environments. Despite its size, it performs well on core NLP tasks.

Grok (xAI) is designed for real-time, conversational interaction and current-event awareness. While its technical details remain limited, it reflects a trend toward domain-specific, responsive LLMs.

Recent research extends beyond architecture to improve scalability, efficiency, and adaptability in LLMs:

MAS-GPT [1] frames multi-agent system construction as a language modeling task, enabling the generation of agent configurations from natural language prompts to reduce manual design and inference cost.

LongRoPE [2] extends context lengths to over 2 million tokens through positional interpolation and staged fine-tuning, enhancing long-document comprehension without architectural overhauls.

CALM [3] introduces modular LLM composition by integrating general-purpose models with smaller, task-specific components via cross-attention, allowing new capabilities while preserving core functionality.

AI-Augmented Predictions [4] demonstrates that pairing human forecasters with LLMs significantly improves decision-making accuracy, highlighting the benefits of human-AI collaboration.

Two-Phase Pretraining [5] investigates the role of data curation and ordering, showing that structured, phased pre-

training can yield up to 17% accuracy improvements when scaled to large datasets.

These advances reflect a growing emphasis on expanding LLM capabilities, improving training efficiency, and enabling human-AI synergy. While this project focuses on building a lightweight LLM from scratch for foundational understanding, it serves as a stepping stone toward future exploration in areas such as long-context processing, modular architectures, and data-centric optimization.

III. Methods

The development of LLM-Lite follows a modular pipeline comprising dataset preparation, tokenization, data loading, model architecture design, hyperparameter optimization, and final retraining. The following subsections describe each component in detail.

A. Dataset Preparation

The dataset is partitioned into training (80%), validation (10%), and testing (10%) subsets. Consistent preprocessing is applied across all splits to ensure compatibility with the tokenizer and model architecture.

B. Custom Byte Pair Encoding (BPE) Tokenizer

A custom Byte Pair Encoding (BPE) tokenizer is implemented to convert raw text into subword-level token sequences. The tokenizer learns frequently co-occurring character pairs from the training set and constructs a vocabulary accordingly. Once trained, the tokenizer is applied uniformly across all dataset splits to maintain consistency and minimize out-of-vocabulary issues.

C. Data Loader with Epoch-wise Shuffling

PyTorch `DataLoader` modules are employed to generate batches for the training, validation, and testing phases. To promote data diversity and mitigate overfitting, the training loader is configured to shuffle the dataset at the beginning of each epoch. Each batch contains tokenized input sequences and their corresponding next-token targets, preprocessed to a fixed block size.

D. Transformer Architecture

The model adopts a decoder-only Transformer architecture inspired by GPT. Key architectural components, such as the number of layers, embedding dimensions, and attention heads, are fully configurable to support scaling based on task complexity. The model can optionally return self-attention weights for interpretability during analysis. Training is performed in an autoregressive manner using a cross-entropy loss function computed between predicted and actual tokens. This architecture adheres to standard Transformer design principles and is optimized for autoregressive sequence modeling.

E. Hyperparameter Search

A structured hyperparameter search was conducted to evaluate various configurations, including learning rates, model depth, attention heads, and embedding dimensions. Each model was trained for 10 epochs, and a custom `hyperparameter_search()` function was used to automate evaluation across configurations. Key metrics—including training loss, validation loss, perplexity, learning rate, and gradient norm—were logged at each epoch for systematic comparison. Perplexity served as the primary evaluation metric, quantifying the model's predictive uncertainty.

The search integrates several training enhancements as follows:

Early stopping. Training is terminated if the validation loss fails to improve over a fixed number of epochs, helping to prevent overfitting and reduce unnecessary computation.

Gradient accumulation. To simulate larger batch sizes under limited GPU memory, gradients are accumulated across multiple steps before performing an optimizer update.

Mixed precision training. Automatic Mixed Precision (AMP) with gradient scaling is employed to accelerate training and improve numerical stability.

Learning rate scheduling. A combination of linear warmup and cosine annealing is used to promote stable convergence and prevent abrupt learning rate changes.

Step-wise tracking. Learning rate values and gradient norms are logged at each optimizer step to support performance analysis and debugging.

Checkpointing. The model achieving the lowest validation perplexity is automatically saved during training to ensure reproducibility and enable recovery if interrupted.

The best-performing configuration identified through this search used a learning rate of 0.001, six transformer layers, four attention heads, and an embedding dimension of 128, achieving a final perplexity of 1.17.

F. Retraining with Optimal Configuration

Following the hyperparameter search, the best-performing configuration was used to retrain the model from scratch. Instead of the original 10 epochs, the retraining process was extended to 20 epochs to investigate whether prolonged training could further enhance performance. Final evaluation was conducted on a held-out test set to assess the model's generalization capability and confirm its overall performance.

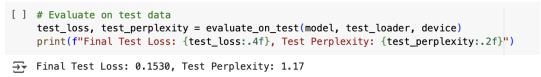


Fig. 1: Model performance on test data

Figure 1 presents the performance of the retrained model on the test data. The resulting test loss is 0.1530, and the corresponding test perplexity is 1.17—comparable to the best configuration obtained from the initial training using the `hyperparameter_search` function. This suggests that

the model may have already reached optimal performance after 10 epochs, likely due to the limited size of the dataset. As a result, further increasing the number of training epochs yields only marginal improvement and may even lead to overfitting.

IV. Experiments

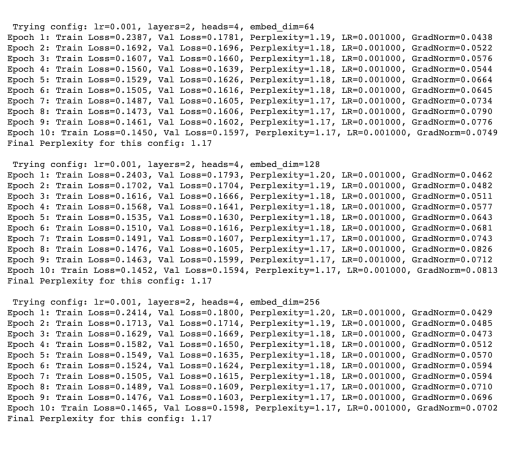
All experiments were conducted on Google Colab using an NVIDIA A100 GPU, a widely accessible platform suitable for researchers without dedicated hardware.

A. Hyperparameter Tuning

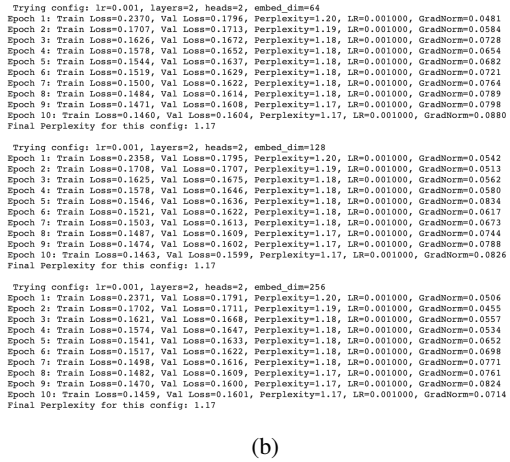
To evaluate the effects of architectural and training hyperparameters on model performance, a grid search was initially performed over the following configuration space:

- **Learning rates:** {0.001, 0.0005}
- **Number of layers:** {2, 4, 6}
- **Attention heads:** {2, 4}
- **Embedding dimensions:** {64, 128, 256}

Initial experiments used a batch size of 64 and trained each configuration for 10 epochs. These runs often required over 10 hours per configuration, occasionally leading to Colab session timeouts. Moreover, performance improvements across configurations were marginal, with perplexity values generally converging around 1.17.



(a)



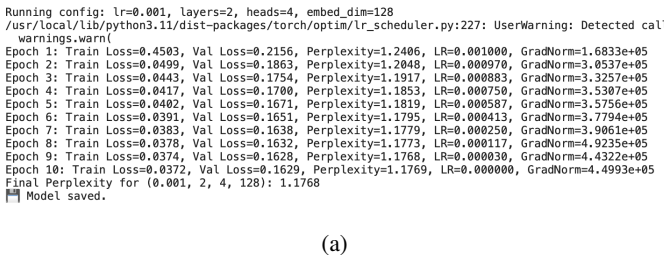
(b)

Fig. 2: Model performance under two configurations.
(a) Configuration A; (b) Configuration B.

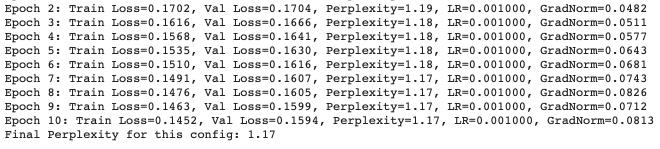
As shown in Figure 2, variations in embedding dimension had minimal impact on final perplexity compared to the number of layers and attention heads. Both configurations (a) and (b), despite differing in embedding sizes, exhibit similar perplexity trends across epochs. This suggests that when model depth and attention heads are limited (i.e., 2 to 4), increasing the embedding dimension does not significantly benefit performance.

To simplify the hyperparameter search and reduce computational cost, the embedding dimension was fixed at 128 in all subsequent experiments. Additionally, the batch size was increased to 128 to improve training throughput. The revised search space was defined as follows:

- **Learning rates:** {0.001, 0.0005}
- **Number of layers:** {2, 4, 6}
- **Attention heads:** {2, 4}
- **Embedding dimension:** 128



(a)



(b)

Fig. 3: Effect of batch size and learning rate on training performance.
(a) Configuration A; (b) Configuration B.

Figure 3 illustrates how changes in batch size and learning rate scheduling affect model performance. Configuration (a) employs a batch size of 128 along with a cosine learning rate scheduler with warmup, allowing the learning rate to decay dynamically toward an optimal value. In contrast, configuration (b) uses a smaller batch size of 64 and a fixed learning rate.

Although configuration (a) begins with higher perplexity than (b), it converges more quickly to a lower perplexity in fewer epochs. This indicates that smaller batch sizes may hinder generalization, particularly with limited training data. Under such conditions, careful tuning of the learning rate becomes even more critical.

```
Running config: lr=0.0005, layers=2, heads=2, embed_dim=128
Epoch 1: Train Loss=0.5783, Val Loss=0.2283, Perplexity=1.2565, LR=0.000500, GradNorm=8.2569e+04
Epoch 2: Train Loss=0.0538, Val Loss=0.2002, Perplexity=1.2217, LR=0.000485, GradNorm=1.7270e+05
Epoch 3: Train Loss=0.0479, Val Loss=0.1852, Perplexity=1.2034, LR=0.000442, GradNorm=2.1066e+05
Epoch 4: Train Loss=0.0448, Val Loss=0.1787, Perplexity=1.1957, LR=0.000375, GradNorm=3.8557e+05
Epoch 5: Train Loss=0.0431, Val Loss=0.1750, Perplexity=1.1913, LR=0.000293, GradNorm=2.2615e+05
Epoch 6: Train Loss=0.0420, Val Loss=0.1726, Perplexity=1.1884, LR=0.000207, GradNorm=4.8730e+05
Epoch 7: Train Loss=0.0412, Val Loss=0.1712, Perplexity=1.1867, LR=0.000125, GradNorm=1.0778e+06
Epoch 8: Train Loss=0.0407, Val Loss=0.1705, Perplexity=1.1859, LR=0.000058, GradNorm=5.6699e+05
Epoch 9: Train Loss=0.0405, Val Loss=0.1702, Perplexity=1.1856, LR=0.000015, GradNorm=5.3871e+05
Epoch 10: Train Loss=0.0403, Val Loss=0.1701, Perplexity=1.1855, LR=0.000000, GradNorm=5.2325e+05
Final Perplexity for (0.0005, 2, 2, 128): 1.1855

Running config: lr=0.0005, layers=2, heads=4, embed_dim=128
Epoch 1: Train Loss=0.6057, Val Loss=0.2256, Perplexity=1.2531, LR=0.000500, GradNorm=1.7581e+05
Epoch 2: Train Loss=0.0536, Val Loss=0.2013, Perplexity=1.2229, LR=0.000485, GradNorm=1.6126e+05
Epoch 3: Train Loss=0.0482, Val Loss=0.1864, Perplexity=1.2049, LR=0.000442, GradNorm=3.6878e+05
Epoch 4: Train Loss=0.0450, Val Loss=0.1794, Perplexity=1.1965, LR=0.000375, GradNorm=4.3047e+05
Epoch 5: Train Loss=0.0432, Val Loss=0.1752, Perplexity=1.1915, LR=0.000293, GradNorm=4.4761e+05
Epoch 6: Train Loss=0.0421, Val Loss=0.1730, Perplexity=1.1889, LR=0.000207, GradNorm=5.2696e+05
Epoch 7: Train Loss=0.0413, Val Loss=0.1715, Perplexity=1.1871, LR=0.000125, GradNorm=5.0048e+05
Epoch 8: Train Loss=0.0408, Val Loss=0.1707, Perplexity=1.1862, LR=0.000058, GradNorm=5.0446e+05
Epoch 9: Train Loss=0.0405, Val Loss=0.1704, Perplexity=1.1858, LR=0.000015, GradNorm=5.2547e+05
Epoch 10: Train Loss=0.0404, Val Loss=0.1704, Perplexity=1.1858, LR=0.000000, GradNorm=5.7078e+05
Final Perplexity for (0.0005, 2, 4, 128): 1.1858
```

Fig. 4: Performance degradation when using a smaller learning rate (0.0005).

Figure 4 shows that a smaller learning rate (0.0005) results in slightly worse performance compared to 0.001. Since the `hyperparameter_search()` function is limited to 10 epochs, smaller initial learning rates may cause weight updates to be too small for effective convergence within the allotted time.

```
Running config: lr=0.001, layers=6, heads=2, embed_dim=128
Epoch 1: Train Loss=0.4192, Val Loss=0.2130, Perplexity=1.2373, LR=0.001000, GradNorm=1.7155e+05
Epoch 2: Train Loss=0.0487, Val Loss=0.1816, Perplexity=1.1991, LR=0.000970, GradNorm=1.7745e+05
Epoch 3: Train Loss=0.0429, Val Loss=0.1714, Perplexity=1.1870, LR=0.000883, GradNorm=9.3522e+04
Epoch 4: Train Loss=0.0404, Val Loss=0.1663, Perplexity=1.1809, LR=0.000750, GradNorm=1.7699e+05
Epoch 5: Train Loss=0.0388, Val Loss=0.1630, Perplexity=1.1771, LR=0.000587, GradNorm=4.6963e+05
Epoch 6: Train Loss=0.0376, Val Loss=0.1609, Perplexity=1.1746, LR=0.000413, GradNorm=1.9082e+05
Epoch 7: Train Loss=0.0367, Val Loss=0.1595, Perplexity=1.1729, LR=0.000250, GradNorm=2.0783e+05
Epoch 8: Train Loss=0.0360, Val Loss=0.1587, Perplexity=1.1720, LR=0.000117, GradNorm=2.4332e+05
Epoch 9: Train Loss=0.0355, Val Loss=0.1583, Perplexity=1.1715, LR=0.000030, GradNorm=2.5831e+05
Epoch 10: Train Loss=0.0352, Val Loss=0.1584, Perplexity=1.1716, LR=0.000000, GradNorm=2.6975e+05
Final Perplexity for (0.001, 6, 2, 128): 1.1715
💾 Model saved.

Running config: lr=0.001, layers=6, heads=4, embed_dim=128
Epoch 1: Train Loss=0.4380, Val Loss=0.2104, Perplexity=1.2342, LR=0.001000, GradNorm=8.3732e+04
Epoch 2: Train Loss=0.0485, Val Loss=0.1812, Perplexity=1.1987, LR=0.000970, GradNorm=3.7457e+05
Epoch 3: Train Loss=0.0428, Val Loss=0.1705, Perplexity=1.1859, LR=0.000883, GradNorm=1.7460e+05
Epoch 4: Train Loss=0.0402, Val Loss=0.1657, Perplexity=1.1803, LR=0.000750, GradNorm=1.9079e+05
Epoch 5: Train Loss=0.0386, Val Loss=0.1624, Perplexity=1.1764, LR=0.000587, GradNorm=1.8960e+05
Epoch 6: Train Loss=0.0374, Val Loss=0.1601, Perplexity=1.1737, LR=0.000413, GradNorm=2.1357e+05
Epoch 7: Train Loss=0.0364, Val Loss=0.1588, Perplexity=1.1721, LR=0.000250, GradNorm=2.1783e+05
Epoch 8: Train Loss=0.0357, Val Loss=0.1581, Perplexity=1.1713, LR=0.000117, GradNorm=2.4464e+05
Epoch 9: Train Loss=0.0352, Val Loss=0.1579, Perplexity=1.1710, LR=0.000030, GradNorm=2.4066e+05
Epoch 10: Train Loss=0.0349, Val Loss=0.1579, Perplexity=1.1711, LR=0.000000, GradNorm=4.8172e+05
Final Perplexity for (0.001, 6, 4, 128): 1.1710
💾 Model saved.
```

Fig. 5: Performance improves when the number of layers exceeds four.

Another key finding, shown in Figure 5, is that increasing the number of attention heads is beneficial only when the model has a sufficient number of layers (at least four). When the depth is too shallow, adding attention heads does not noticeably improve performance.

Final perplexity values across all tested configurations fell within a narrow range of 1.1858 to 1.1710, reflecting consistent and stable convergence across the grid search.

B. Text Generation Example

A sample text generation test was performed using the prompt `"The university student"`. The model generated an extended continuation, followed by a filtering step to remove repeated tokens. The resulting output—`"The university student Howard organization."` Demonstrates the model's ability to maintain thematic relevance while introducing new content. This result suggests that prompt-driven generation, when paired with simple post-processing, can produce coherent and semantically aligned outputs.

C. Prompt Sensitivity and Sampling Strategy Analysis

The model exhibits strong sensitivity to minor changes in prompt phrasing. Small variations—such as modifying ``` ``The university student'' ``` to ``` ``A university student'' ```—produce significantly different outputs, often with excessive repetition or reduced coherence. For instance, Prompt 2 repeats the word `student`, while Prompt 7 overuses `education`-related tokens. Pairwise BLEU scores further support this, with most values below 0.05, reflecting low lexical similarity across prompts.

Sampling strategy also plays a critical role. A low temperature (0.7) yields more deterministic but shorter responses, while higher top-k values increase diversity but can reduce coherence. These findings highlight the model's sensitivity to prompt formulation and the importance of carefully selecting decoding parameters.

D. Head Ablation Experiment

This experiment investigates the effect of selectively disabling specific self-attention heads during text generation. A head mask is applied to a decoder-only Transformer model with four attention heads. In this setting, heads 1 and 2 (indexed from 0) are ablated by setting their corresponding mask values to 0, while the remaining heads remain active.

The model is prompted with the input:

```
The cat sat on
```

The generated continuation is:

```
The cat sat on on on Mul on Mul on
Mul on Mul Mul Mul on Mul Mul on
Mul Mul Mul
```

This output demonstrates degraded fluency and coherence, as it exhibits repetitive and semantically inconsistent tokens. The excessive repetition of phrases such as `on Mul` and `Mul Mul` suggests that the ablated heads play an important role in maintaining contextual diversity and regulating repetition.

The experiment highlights how specific attention heads contribute to different aspects of language modeling, and how their removal can disrupt the balance between fluency, diversity, and semantic structure in generated text.

E. Attention Map Analysis

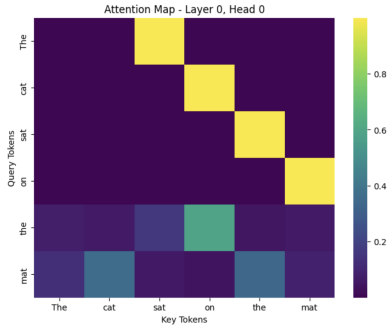


Fig. 6: Attention map on Layer 0, Head 0

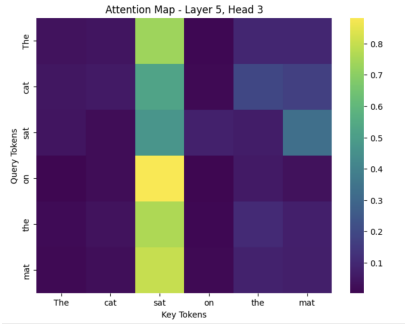


Fig. 7: Attention map on Layer 5, Head 3

Attention maps were visualized across different layers and attention head configurations. When the layer is set to 0, as shown in Figure 6, the model exhibits strong local attention, primarily focusing on the immediately preceding token. This indicates that the initial layer has limited ability to capture long-range dependencies. As the number of attention heads increases, the attention becomes more distributed, as shown in Figure 7, enabling the model to attend to a broader range of tokens within the sequence. Additionally, in deeper layers, the model begins to focus on key tokens—such as the verb *sat*—which are crucial for understanding sentence structure and enhancing next-token prediction.

V. Visualization Results

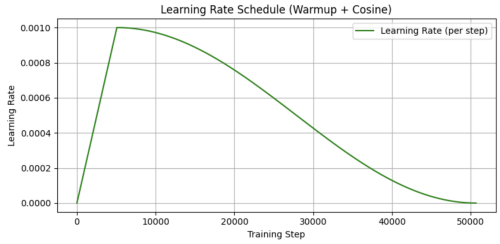


Fig. 8: Learning rate schedule using a warmup phase followed by cosine decay.

Figure 8 shows the learning rate schedule applied during training, combining a warmup phase with cosine decay. The learning rate increases linearly from zero to a peak of 0.001 over the first 8000 steps, helping to stabilize early training and prevent gradient explosion. It then decays smoothly following a cosine pattern, tapering toward zero by step 50,000. This schedule facilitates rapid initial learning and controlled convergence in later stages.

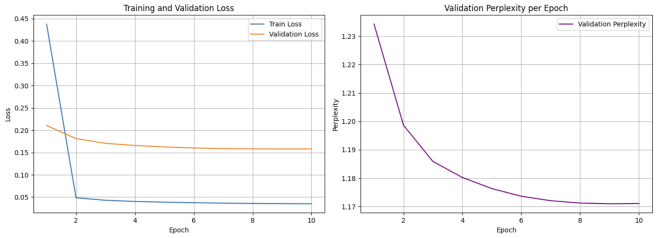


Fig. 9: Training and validation loss (left) and validation perplexity (right) over 10 epochs.

Figure 9 displays the training dynamics over 10 epochs. On the left, the training loss drops sharply in the early epochs and quickly stabilizes at a low level, reflecting effective learning. The validation loss steadily decreases and flattens around epoch 7, indicating stable generalization without overfitting. The right subplot shows validation perplexity decreasing from approximately 1.234 to 1.171, suggesting improved model confidence and predictive capability. Together, the trends confirm a smooth and well-converged training process.

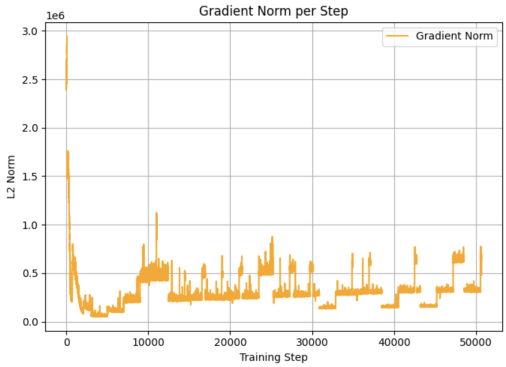


Fig. 10: L2 gradient norm per training step throughout the training process.

Figure 10 plots the L2 norm of gradients at each training step. Early training shows large and volatile gradients—typical during warmup as the model adjusts from random initialization. As training continues, the gradients stabilize with moderate fluctuations. Occasional spikes suggest larger updates due to difficult samples or learning rate transitions. Overall, the gradient behavior remains numerically stable, aided by mixed-precision scaling.

VI. Conclusion

This project presents a lightweight, from-scratch implementation of a large language model (LLM) aimed at deepening understanding of the core components and training dynamics of transformer-based architectures. Built through a structured pipeline—encompassing data preprocessing, tokenizer construction, model training, and empirical evaluation—the model demonstrated strong language modeling performance, achieving low perplexity despite limited computational resources.

Domain-specific expansion. Incorporating datasets from specialized fields such as biology, mathematics, or programming could evaluate the model's ability to generalize across domains while maintaining high performance in focused areas.

Advanced fine-tuning. Future work may integrate parameter-efficient tuning strategies, such as PEFT, and utilize frameworks like the Hugging Face Trainer to streamline adaptation across downstream tasks and reduce training overhead.

Attention head analysis. By systematically varying the number of attention heads while holding other hyperparameters constant, further experiments could reveal how attention span and granularity influence prediction accuracy and model interpretability.

Optimization algorithm comparison. Evaluating the training efficiency and convergence behavior of optimizers such as Adam versus Stochastic Gradient Descent (SGD) could provide insights into the trade-offs between speed, stability, and generalization.

Together, these directions offer meaningful opportunities to build on the current work and advance the understanding and development of efficient, interpretable, and task-adaptable language models.

REFERENCES

[1] R. Ye, S. Tang, R. Ge, Y. Du, Z. Yin, S. Chen, and J. Shao, "MAS-GPT: Training LLMs to Build LLM-based Multi-Agent Systems," *arXiv preprint arXiv:2404.08535*, submitted on 5 Mar. 2025. [Online]. Available: https://arxiv.org/abs/2404.08535

[2] Y. Ding, L. L. Zhang, C. Zhang, Y. Xu, N. Shang, J. Xu, F. Yang, and M. Yang, "LongRoPE: Extending LLM Context Window Beyond 2 Million Tokens," *arXiv preprint arXiv:2402.12821*, submitted on 21 Feb. 2024. [Online]. Available: https://arxiv.org/pdf/2402.13753

[3] R. Bansal, B. Samanta, S. Dalmia, N. Gupta, S. Vashishth, S. Ganapathy, A. Bapna, P. Jain, and P. Talukdar, "LLM-Augmented LLMs: Expanding Capabilities Through Composition," *arXiv preprint arXiv:2402.19118*, submitted on 4 Jan. 2024. [Online]. Available: hhttps://www.prateekjain.org/publications/all_papers/BansalSDGVGBJT24.pdf

[4] P. Schoenegger, P. S. Park, E. Karger, S. Trott, and P. E. Tetlock, "AI-Augmented Predictions: LLM Assistants Improve Human Forecasting Accuracy," *ACM Trans. Interact. Intell. Syst.*, vol. 15, no. 1, Art. 4, Feb. 2025. [Online]. Available: https://dl.acm.org/doi/full/10.1145/3707649

[5] S. Y. Feng, S. Prabhumoye, K. Kong, D. Su, M. Patwary, M. Shoeybi, and B. Catanzaro, "Maximize Your Data's Potential: Enhancing LLM Accuracy with Two-Phase Pretraining," *arXiv preprint arXiv:2403.05291*, submitted on 6 Mar. 2024. [Online]. Available: https://arxiv.org/pdf/2412.15285

[6] A. Singh, "Perplexity for LLM Evaluation," GeeksforGeeks, Jul. 4, 2023. [Online]. Available: https://www.geeksforgeeks.org/perplexity-for-llm-evaluation/

[7] S. Raschka, *Build a Large Language Model (From Scratch)*. New York, NY, USA: Simon and Schuster, Oct. 2024.