GPT-2 Unveiled: Comparative Insights

Peeyush Dyavarashetty
MS in Applied Machine Learning
University of Maryland
College Park, MD 20742
peeyush@umd.edu

Sumedha Vadlamani
MS in Data Science
University of Maryland
College Park, MD 20742
sumedha6@umd.edu

Thilak Cebolu Mohan
MS in Applied Machine Learning
University of Maryland
College Park, MD 20742
thilakcm@umd.edu

Abstract

The purpose of this work is to develop a deeper understanding of GPT-2 by building it from scratch and systematically analyzing techniques that improve its performance. We pretrain a GPT-2 model built from scratch with 124 million parameters on the 10-billion-token FineWeb-Edu dataset, leveraging distributed Data Parallel (DDP) processing. To explore the impact of architectural choices, we integrate a slew of positional encodings and attention mechanisms. Using the HellaSwag benchmark, we assess model accuracy and scalability, while Weights & Biases aids in analyzing attention maps, parameter distributions, and gradient behavior. This work provides insights into performance differences and especially dives deep into understanding what drives performance in GPT-like models. The complete implementation of this project is available on GitHub. **Link to Codebase:** https://github.com/Thilak-cm/GPT2-Stripped-Comparative-Insights

1 Introduction

Large language models (LLMs) have revolutionized natural language processing (NLP), driving advances in tasks such as text generation, translation, and question answering. GPT, developed by OpenAI, has been instrumental in showcasing the potential of autoregressive models to capture complex linguistic patterns and generate coherent text. In this work, our goal is to build a deep understanding of GPT-2 by pretraining a 124-million-parameter model from scratch on the 10-billion-token FineWeb-Edu dataset. Our goals are to:

- Investigate the impact of advanced attention mechanisms and positional encodings on model performance and scalability.
- To identify the key factors within these techniques that contribute to performance improvements.
- Focusing on a 124M parameter baseline model is a huge undertaking, as building production-grade language models requires immense computational resources and exceeds the scope of this project.

To optimize the training process, we utilized Distributed Data Parallel (DDP) processing across A5000 and A6000 GPUs, completing 20,000 epochs efficiently. Two advanced attention mechanisms, Flash

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

Attention and Linformer, were integrated to address the computational inefficiencies of traditional self-attention while maintaining the model's ability to capture long-range dependencies. Additionally, six positional encoding schemes were evaluated: Baseline (Sinusoidal), Learned, RoPE, ALiBi, FIRE, and Kerple to understand their role in encoding sequence order and improving generalization.

The performance of the model was benchmarked in HellaSwag, a challenging data set to test natural language inference and reasoning. We picked HellaSwag because it shows the growth in performance of the model at designated checkpoints in training which gives insights into scalability as well. To gain deeper insights, we used Weights & Biases for analyzing attention maps, parameter distributions, and gradient behaviors. Our findings highlight the influence of efficient attention mechanisms and positional encodings on model performance, providing a foundation for future research into optimizing large-scale language models.

2 Related Work

2.1 Transformer Architecture and GPT-2

The Transformer architecture, introduced by Vaswani et al. (2017), revolutionized natural language processing by using attention mechanisms [9]. This architecture underpins large language models like GPT-2, developed by OpenAI, which has 1.5 billion parameters and was trained on 8 million web pages [7]. In this project, we implement the smallest version of GPT-2 due to resource constraints.

The core innovation in Transformers is the attention mechanism, particularly self-attention, which allows the model to focus on relevant parts of the input sequence [9]. Multi-head attention further enhances this capability by jointly attending to information from different representation subspaces. Since Transformers lack inherent sequential processing, positional encodings (PEs) are used to incorporate word order information. The original Transformer employed sinusoidal positional encodings as a baseline [5]. Subsequent work introduced learned and relative positional encodings, which often outperform sinusoidal PEs, especially for longer sequences [5].

As models have grown, researchers have focused on improving scalability through model parallelism, efficient attention mechanisms like flash attention, and optimized training techniques [2, 9]. GPT-2 and similar models have excelled in tasks like text generation, summarization, and question answering, evaluated using metrics such as perplexity and performance on downstream tasks [7].

3 Method

In this section, we describe the various positional encoding techniques used in GPT-2 and other transformer models, including learned embeddings, sinusoidal encodings, Rotary Positional Embedding (RoPE), Attention with Linear Biases (ALiBi), Functional Interpolation for Relative Positions (FIRE), Kerple, and Linformer.

For all experiments, we implemented a GPT-2 model with 124 million parameters, following the architecture described in the original GPT-2 paper [7]. The model configuration included:

- **Sequence Length (block size):** 1024,
- **Vocabulary Size:** 50257 (we actually use 50304 because it leaves no overflow when divided by 128),
- **Number of Layers:** 12,
- **Number of Attention Heads:** 12,
- **Embedding Dimension:** 768.

We adhered closely to the hyperparameters and training setup from the original GPT-2 paper. Specifically:

- **Optimizer:** AdamW with $\beta_1 = 0.9$, $\beta_2 = 0.95$, and $\epsilon = 10^{-8}$.
- **Learning Rate:** Cosine decay scheduling, starting at 6×10^{-4} and decreasing to 6×10^{-5}, with a linear warmup over 715 steps.

- **Weight Decay:** 0.1.
- **Gradient Clipping:** Maximum norm of 1.0.
- **Batch Size:** A total of 512K tokens, achieved through gradient accumulation across multiple steps, with a per-GPU batch size adjusted to fit available GPU memory.
- **Precision:** Mixed-precision training using bfloat16 with TensorFloat-32 (TF32) enabled for matrix multiplications on compatible hardware.

To evaluate the impact of different positional encodings and attention mechanisms on performance and scalability, we trained the model over 20,000 steps using the FineWeb-Edu dataset consisting of 10 billion tokens. Training was distributed across A5000 and A6000 GPUs using PyTorch's Distributed Data Parallel (DDP) framework.

3.1 Positional Encodings

3.1.1 Learned Positional Embeddings

GPT-2 uses learned positional embeddings [7]. This approach involves initializing a vector of positional embeddings for each position from 0 to the maximum sequence length. These embeddings are then updated during training using gradient descent, allowing the model to learn the most effective representation of position for the given task.

3.1.2 Sinusoidal Positional Encodings

While not used in GPT-2, the original Transformer model introduced sinusoidal positional encodings [9]. These are fixed encodings defined by sine and cosine functions:

$$PE(\text{pos}, 2i) = \sin(\text{pos}/10000^{2i/d_{\text{model}}}) \tag{1}$$

$$PE(\text{pos}, 2i+1) = \cos(\text{pos}/10000^{2i/d_{\text{model}}}) \tag{2}$$

where pos is the position and i is the dimension. This approach allows the model to extrapolate to sequence lengths longer than those seen during training.

3.1.3 RoPE

RoFormer [8] introduces Rotary Position Embedding (RoPE), which encodes positional information using rotation matrices. RoPE combines absolute positional information with explicit relative position dependency in the self-attention mechanism. The key idea is to apply rotation matrices to the query and key vectors based on their absolute positions, enabling the self-attention to incorporate relative positional relationships efficiently. The rotary position embedding for a token at position m can be expressed as:

$$R_{\Theta,\{q,k\}}(x_m, m) = \begin{pmatrix} cosm\theta & -sinm\theta \\ sinm\theta & cosm\theta \end{pmatrix} W_{\{q,k\}} x_m \tag{3}$$

where x_m is the token embedding at position m and θ is a hyperparameter controlling the rate of rotation.

3.1.4 Attention with Linear Biases (ALiBi)

ALiBi adds a penalty to the attention weight scores proportional to the distance between tokens [6]. The attention score between tokens i and j is calculated using the equation 4, where m is a head-specific scalar that is fixed during training.

$$\text{Attention score} = q_i \cdot k_j + m \cdot |i - j| \tag{4}$$

3.1.5 Functional Interpolation for Relative Positions (FIRE)

FIRE uses a learnable continuous function to map input positions to biases [4]. It employs progressive interpolation to ensure bounded input for the position encoding function for all sequence lengths. FIRE positional encoding function with logarithmic transformation and adaptive thresholding:

$$b_{\text{FIRE}}(i, j) = f_\theta \left(\frac{\psi(i-j)}{\psi(\max\{L, i\})} \right) \tag{5}$$

where $\psi(x) = \log(cx + 1)$, $c > 0$ is a learnable parameter and $L > 0$ is a learnable threshold. The MLP structure for f_θ:

$$f_\theta(x) = v_3^T \sigma(V_2 \sigma(v_1 x)) \tag{6}$$

where σ is the ReLU activation function, $v_1, v_3 \in \mathbb{R}^s$, $V_2 \in \mathbb{R}^{s \times s}$ and s is the hidden size of the MLP. Attention calculation incorporating FIRE is similar to ALiBi.

3.1.6 Kerple

The logKerple variant of the KERPLE framework [1] uses a logarithmic function to compute the relative positional embedding. The basic form of the logKerple function is

$$k(m, n) = -r_1 \log(r_2 |m - n| + 1) \tag{7}$$

where r_1 and r_2 are trainable. The full attention score calculation incorporating logKerple

$$a_{m,n} = \frac{\exp\left(q_m^T k_n + k(m, n)\right)}{\sum_{i=1}^{L} \exp\left(q_m^T k_i + k(m, i)\right)} \tag{8}$$

where m and n are token positions with q_m and k_n as query and key vectors respectively, d is the dimension of the embedding space and L is the sequence length. The logKerple function k(m,n) provides a logarithmic decay in the attention scores based on the relative distance between tokens, which allows for better length extrapolation compared to other positional embedding methods.

3.2 Attention mechanisms

3.2.1 Regular Attention

Regular attention in transformer models computes pairwise interactions between all tokens in a sequence, resulting in quadratic time and memory complexity. The standard attention mechanism is defined as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V \tag{9}$$

where Q, K, and V are query, key, and value matrices, respectively, and d_k is the dimension of the key vectors. This operation has a time and space complexity of $\mathcal{O}(n^2 d)$, where n is the sequence length and d is the hidden dimension.

3.2.2 Flash Attention

Flash Attention [2] is an optimized attention algorithm that reduces memory usage and improves computational efficiency. It achieves this by dividing the input sequence into blocks and processing them in a tiled manner, recomputing intermediate values when needed instead of storing them. The core computation remains the same as regular attention, but Flash Attention optimizes the implementation:

$$S = QK^T, \quad P = \text{softmax}(S), \quad O = PV \tag{10}$$

Flash Attention computes these operations in blocks, maintaining running sums for softmax normalization. This approach reduces memory complexity to $\mathcal{O}(n)$ and improves speed by better utilizing GPU memory hierarchy, making it particularly effective for long sequences and large batch sizes.

3.2.3 Efficient Attention

Efficient Attention [3] reduces the computational and memory complexity of self-attention from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$, where N is the sequence length. This is achieved by expressing self-attention as a linear dot-product of kernel feature maps using $\phi(x) = \text{elu}(x) + 1$, ensuring positive similarity scores. The approach leverages the associativity property of matrix products to accelerate computation and reveals a connection to recurrent neural networks. However, this method primarily targets the encoder and does not directly address causal masking challenges in autoregressive decoders, which are essential for preventing information leakage from future tokens.

3.2.4 Linformer

Linformer is a linear self-attention mechanism designed to reduce the computational complexity of transformers [10]. It approximates the self-attention mechanism using the factorization of the low-rank matrix, reducing the total complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$ in both time and space, where N is the sequence length. The Linformer approximates the attention mechanism by projecting the key and value matrices to a lower-dimensional space.

$$\text{LinformerAttention}(Q, K, V) = \text{softmax}\left(\frac{Q(E_K K)^T}{\sqrt{d}}\right)(E_V V) \tag{11}$$

where: $E_K \in \mathbb{R}^{k \times n}$ and $E_V \in \mathbb{R}^{k \times n}$ are learned projection matrices, k is the projected dimension ($k << n$) and n is the sequence length.

4 Results

In this section, we present a comprehensive analysis of the experiments conducted to evaluate the impact of attention mechanisms and positional encodings on model performance and scalability. We explore results across different configurations, including training and testing performance on 2 GPUs and 4 GPUs, the effects of varying batch sizes, and a direct comparison between Flash Attention and Regular Attention. Key evaluation metrics such as train loss, validation loss, HellaSwag accuracy, and tokens per second are used to assess performance.

To provide clarity and coherence, the results are organized into the following subsections:

1. Positional Encodings – Detailed comparisons of six encodings: Sinusoidal, RoPE, ALiBi, FIRE, Kerple, and Learned PE.
2. Attention Mechanisms – Evaluation of Flash, Linformer, and Regular Attention.

4.1 Comparison Across Positional Encodings

To effectively compare the performance of various positional encodings (PEs), we analyze four key metrics: *Best Training Loss*, *Validation Loss*, *Average Tokens per Second*, and *HellaSwag Accuracy*. Figures 1, 2, 3, and 11d illustrate these metrics.

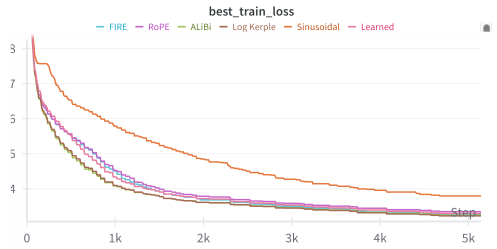


Figure 1: Best Training Loss

Figure 2: Validation Loss

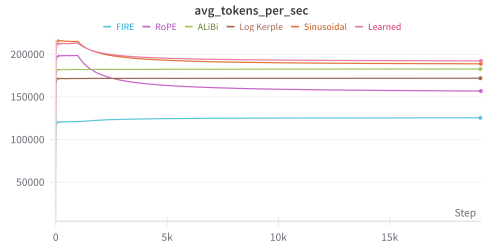


Figure 3: Average Tokens per Second

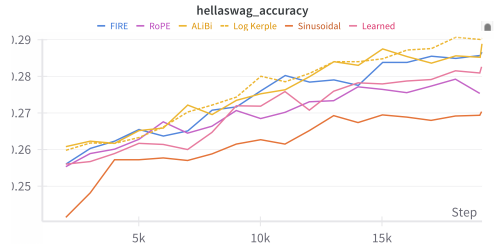


Figure 4: HellaSwag Accuracy

The *train loss* plot (Figure 1) illustrates convergence behavior during training. Sinusoidal PE lags behind the rest, displaying slower convergence and higher loss throughout. In contrast, Learned,

RoPE, ALiBi, and Log Kerple converge rapidly, achieving similar training losses after sufficient steps. The *val_loss* plot (Figure 2) highlights model performance on the validation set. Sinusoidal PE consistently exhibits higher loss compared to other encodings, indicating weaker generalization. Learned, RoPE, and ALiBi achieve the lowest validation losses, reinforcing their superior performance. The *avg_tokens_per_sec* metric (Figure 3) evaluates training efficiency. Learned PE achieves the highest throughput, followed by Sinusoidal and RoPE. ALiBi and Log Kerple lag significantly behind, highlighting their computational overhead during training. The *hellaswag_accuracy* plot (Figure 11d) compares model performance during testing. Sinusoidal PE underperforms notably, achieving the lowest accuracy. Learned and RoPE lead in performance, while ALiBi and Log Kerple demonstrate competitive but slightly lower results.

4.1.1 Discussion

Sinusoidal positional encoding consistently underperforms across all metrics—training loss, HellaSwag accuracy, and throughput. To understand why, we analyze the nature of learned positional embeddings and compare them to the sinusoidal embeddings used in the original Vaswani et al. [9] implementation. Figure 5 visualizes specific dimensions of the learned positional embeddings matrix. Interestingly, the learned embeddings exhibit sinusoidal-like patterns, especially at initial positions, where we observe higher variance and frequencies. This suggests that the model inherently optimizes towards sinusoidal patterns for certain dimensions. However, the learned curves are not perfect; they appear slightly jagged and, upon closer inspection, reveal significant warping and deviations from ideal sinusoidal forms. This jaggedness indicates that while sinusoidal embeddings provide a reasonable approximation of positional information, fully learned embeddings can adapt and optimize for specific tasks or datasets. The original sinusoidal implementation, though efficient, lacks the flexibility of learned embeddings to fine-tune positional information.

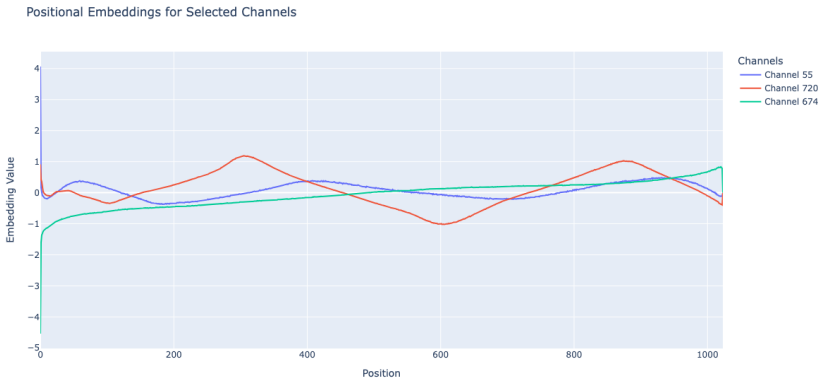


Figure 5: Visualization of selected channels from the learned positional embedding matrix, highlighting sinusoidal-like patterns.

4.2 Comparison Across Attention Mechanisms

We evaluated *Flash Attention*, *Linformer* (with $K = 256$ and $K = 216$), and *Regular Attention* across training and testing phases. Key metrics include training loss, average tokens per second, HellaSwag accuracy, and validation loss. The *best train loss* and *avg_tokens_per_sec* plots (Figures 6 and 6) reveal distinct trends. *Flash Attention* achieves significantly faster convergence, stabilizing at a much lower training loss compared to Linformer and Regular Attention. This is attributed to its memory efficiency and reduced computational overhead. However, the speed improvements come at a slight cost. While *Flash Attention* ingests tokens at an impressive rate exceeding 200K tokens/sec, *Regular Attention* falls behind at 160K tokens/sec, and *Linformer* lags further at under 100K tokens/sec for both $K = 216$ and $K = 256$ configurations.

The testing phase highlights the trade-offs between performance and efficiency. *HellaSwag accuracy* (Figure 7) shows that *Flash Attention* consistently outperforms Linformer and Regular Attention, achieving nearly 0.28 accuracy, whereas *Linformer* struggles to surpass 0.26. Similarly, the *validation*

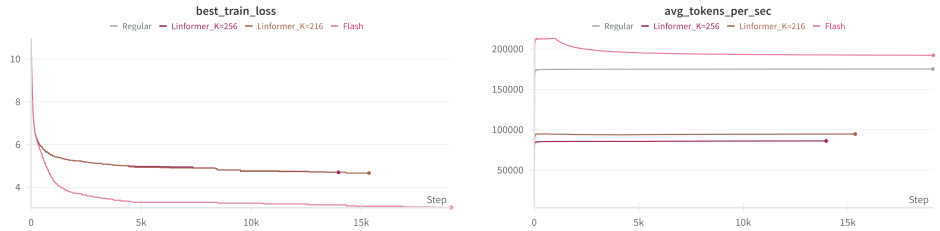


Figure 6: Training performance: Best train loss (left) and average tokens per second (right) across attention mechanisms.

loss plot (Figure 7) further validates the effectiveness of Flash Attention, as it maintains a significantly lower loss throughout testing. This reinforces that Flash Attention balances both efficiency and performance, unlike Linformer, which compromises accuracy for computational benefits.

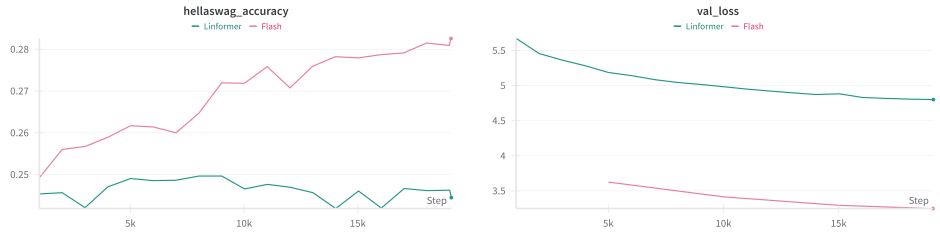


Figure 7: Testing performance: HellaSwag accuracy (left) and validation loss (right) across attention mechanisms.

4.2.1 Discussion

The results clearly indicate that *Flash Attention* delivers superior performance and scalability, outperforming Linformer and Regular Attention in both training and testing phases. The reduced computational complexity of Flash Attention enables faster training (higher throughput) while maintaining accuracy, making it an ideal choice for resource-constrained environments.

In contrast, *Linformer*—while computationally efficient—fails to achieve comparable accuracy. Its approximations result in higher validation loss and lower downstream task performance, as evidenced by the HellaSwag accuracy.

4.3 Comparison Across Batch Sizes

The impact of batch size on RoPE-based models is analyzed across three metrics: training loss, average tokens per second, and HellaSwag accuracy. Each metric is visualized for batch sizes $B = 16$ and $B = 32$, providing insights into convergence, efficiency, and performance.

Figure 8: Training loss comparison for RoPE across $B = 16$ and $B = 32$.

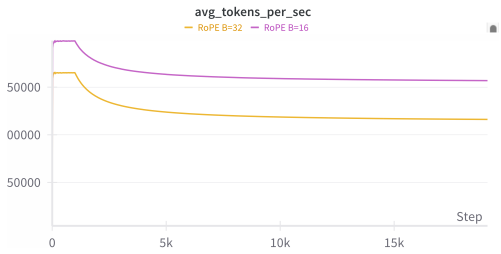


Figure 9: Average tokens per second for RoPE across $B = 16$ and $B = 32$.

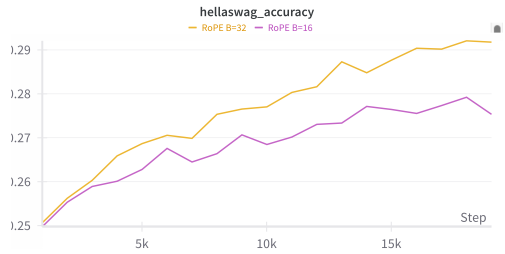


Figure 10: HellaSwag accuracy comparison for RoPE across $B = 16$ and $B = 32$.

The *avg_tokens_per_sec* metric (Figure 9) highlights the efficiency differences between batch sizes. $B = 32$ exhibits lower throughput compared to $B = 16$, which is expected due to the computational overhead associated with larger batches. The *hellaswag_accuracy* plot (Figure 10) demonstrates that $B = 32$ consistently outperforms $B = 16$ during testing. This trend indicates that larger batch sizes may contribute to better generalization on downstream tasks, despite minimal differences in training loss.

4.3.1 Discussion

The observed trends were consistent across different positional encodings; for brevity, we present results only for RoPE. Despite a slower token ingestion rate at larger batch sizes (Figure 9), we observe in Figure 10 that higher batch sizes (B=32) lead to improved scalability and better performance. Interestingly, B=32 took approximately 5 extra hours to complete 20k epochs compared to B=16, revealing a trade-off between longer training times and enhanced performance. This raises the question: why not train B=16 for a longer duration? While we haven't explicitly tested this, it is likely that B=32 will still outperform B=16 even with extended training. Larger batch sizes offer smoother gradient updates, reduced variance, and better utilization of hardware, leading to faster convergence and improved optimization in the long run.

5 Conclusion

In this work, we pretrained a GPT-2 model with 124M parameters from scratch, integrating various positional encodings (PEs) and attention mechanisms to analyze their impact on model performance and efficiency. Through extensive experiments conducted on the FineWeb-Edu dataset, we systematically evaluated **six positional encodings**—Learned, Sinusoidal, RoPE, ALiBi, FIRE, and Kerple—alongside **three attention mechanisms**—Regular, Flash, and Linformer.

Our results show that **Flash Attention** consistently outperforms other mechanisms in terms of convergence speed, throughput, and downstream task accuracy while maintaining efficiency. Among the positional encodings, **RoPE** and **Learned PEs** emerged as the strongest performers, achieving faster convergence and better generalization. However, these gains often come at a computational cost, as seen in **FIRE**, which trades higher parameter count for slower throughput.

We also explored the impact of batch size, demonstrating that larger batches (**B=32**) improve scalability but increase training time. Clever workarounds, such as splitting long runs and logging artifacts remotely, enabled us to overcome computational constraints.

This project not only highlights the trade-offs between performance and efficiency across different architectural enhancements but also deepens our understanding of the underlying mechanisms that drive improvements in large-scale transformer models. Future work includes scaling these findings to larger models and datasets to validate the observed trends further.

References

[1] Ta-Chung Chi, Ting-Han Fan, Peter J. Ramadge, and Alexander I. Rudnicky. Kerple: Kernelized relative positional embedding for length extrapolation, 2022.

[2] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[3] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention, 2020.

[4] Shanda Li, Chong You, Guru Guruganesh, Joshua Ainslie, Santiago Ontanon, Manzil Zaheer, Sumit Sanghai, Yiming Yang, Sanjiv Kumar, and Srinadh Bhojanapalli. Functional interpolation for relative positions improves long context transformers, 2024.

[5] Taro Miyazaki, Hideya Mino, and Hiroyuki Kaneko. Understanding how positional encodings work in transformer model. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 17011–17018, 2024.

[6] Ofir Press, Noah A. Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation, 2022.

[7] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[8] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023.

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[10] Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity, 2020.

A Hardware and Resource Details

Our training experiments were conducted on two separate clusters, each with unique hardware configurations:

- **Cluster 1**: Single node with **4 NVIDIA A5000 GPUs** (24GB RAM each) — utilized for experiments with batch size $B = 16$.
- **Cluster 2**: Single node with **2 NVIDIA A6000 GPUs** (50GB RAM each) — utilized for experiments with batch size $B = 32$.

These hardware differences allowed us to compare the efficiency of training at varying batch sizes while maintaining consistency across other hyperparameters.

B Supplementary Results

This section presents additional experimental results and analyses that complement the findings discussed in the main paper. These results provide further insights into the performance, efficiency, and behavior of various positional encodings (PEs) and attention mechanisms evaluated in our work. While not included in the main text due to space constraints, they serve to validate and reinforce our conclusions.

We include the following:

- Results from additional experiments across multiple configurations and metrics.
- Comparisons and ablation studies that explore the trade-offs in computational efficiency, throughput, and performance.
- Plots and analyses that demonstrate trends observed during model training and evaluation.

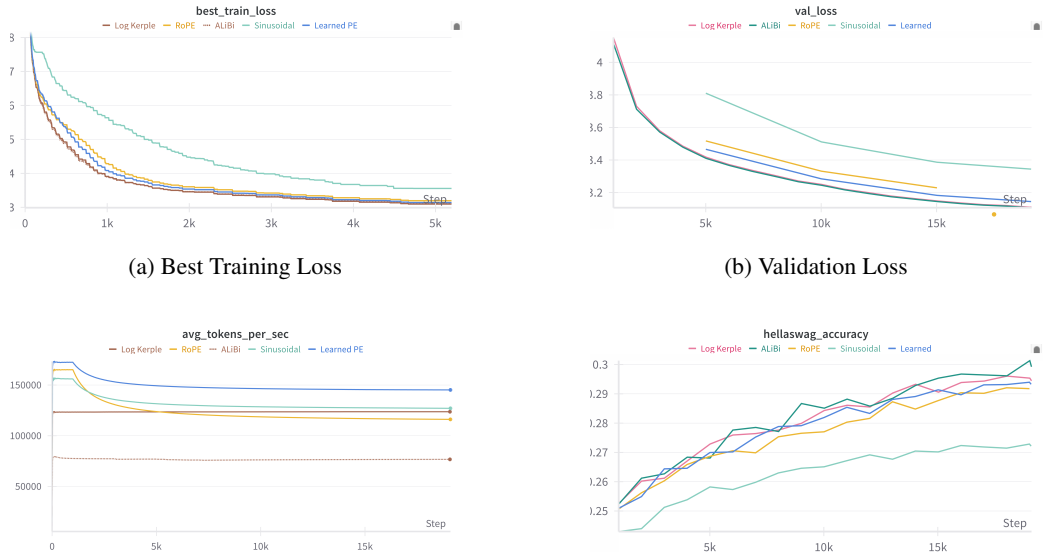
B.1 Comparison Across Positional Encodings when B=32

In the results section earlier, we compared the performance of various positional encodings (PEs) for B=16 . Here, we present the same comparison for B=32 . However, FIRE is excluded due to its high

memory requirements, which caused frequent memory issues stemming from its larger parameter count.

Notably, Sinusoidal PE exhibits a plateau in training loss between approximately 100 to 250 epochs. This flattening is likely due to its rigid, predefined nature, which limits adaptability during training. Additionally, its fixed structure restricts gradient flow and introduces high variance at initial positions, leading to temporary stagnation. In contrast, learned positional encodings dynamically adapt to the data, enabling smoother optimization and faster convergence, thereby avoiding such plateaus.

Figures 11a, 11b, 11c, and 11d illustrate these metrics.



(a) Best Training Loss

(b) Validation Loss

(c) Average Tokens per Second

(d) HellaSwag Accuracy

Figure 11: Performance comparison across positional encodings with $B = 32$.

B.2 Comparison Across Different Attention Mechanisms

In this subsection, we compare the performance of FIRE Attention with its two variants: **FIRE Regular** and **FIRE Flash**. The experiments were conducted with a batch size of $B = 8$, and the evaluation metrics include **training loss** and **average tokens per second (throughput)**. The results highlight the differences in computational efficiency and convergence behavior between these attention variants.

B.2.1 Training Performance

Figure 12 compares the training loss across **FIRE Regular** and **FIRE Flash**. Both variants exhibit similar convergence trends. However, FIRE Flash achieves a marginally faster drop in training loss early in the training process. This can be attributed to the computational efficiency of Flash Attention, which allows for more stable and efficient updates to the model parameters.

B.2.2 Throughput Efficiency

Throughput, measured as **average tokens per second**, highlights the computational advantages of FIRE Flash over FIRE Regular. As shown in Figure 13, FIRE Flash consistently processes tokens at a higher rate compared to FIRE Regular. This is a direct result of the optimized implementation in Flash Attention, which reduces computational overhead and improves memory efficiency during training.

Figure 12: Training loss comparison between FIRE Regular and FIRE Flash with $B = 8$.

Figure 13: Throughput comparison between FIRE Regular and FIRE Flash with $B = 8$.

B.2.3 Key Observations

- **FIRE Flash** outperforms FIRE Regular in terms of throughput, achieving higher tokens per second due to its efficient memory utilization.
- Both variants exhibit similar convergence behavior, but FIRE Flash achieves a slight edge in early-stage training, converging faster due to optimized computational efficiency.
- Despite these advantages, the memory footprint of FIRE remains a limiting factor for larger batch sizes or longer sequence lengths, as observed in earlier experiments.

In summary, **FIRE Flash** demonstrates clear benefits in terms of speed and throughput compared to FIRE Regular. These results suggest that combining FIRE with Flash Attention can provide substantial computational efficiency improvements while maintaining strong convergence performance.

C Resource Limitations and Workarounds

Throughout the experiments, we encountered several challenges due to resource constraints:

- **Memory Issues**: Each user account on the cluster was allotted a maximum of **30GB of memory**. This limited our ability to cache large training files locally. As a workaround:
  - All logs, model checkpoints, and artifacts were uploaded to **Weights & Biases (W&B)** for seamless logging and retrieval.
  - Checkpoints were saved periodically to allow resumption of training in case of interruptions.
- **Runtime Limits**: The maximum allowed runtime per job on our Nexus account was **2 days**. For runs that exceeded this time cap:
  - Training was split into multiple sessions.
  - We resumed subsequent sessions by loading the last saved model weights (`.pth` files), effectively continuing from where the previous session left off.

These strategies ensured that training remained uninterrupted while staying within the cluster's constraints.

C.1 Comparison of Trainable Parameters Across Models

Figure 14 shows the comparison of trainable parameters for the models considered in our work. The models utilizing **Learned PE** and **RoPE** were excluded from this comparison due to their substantially higher parameter counts, which result from additional learnable embeddings and transformations.

Among the remaining models, the **FIRE** model exhibits the highest parameter count. This increase in parameters likely contributes to its slower response times when used in inference scenarios, such as our chatbot application built from the saved weights.

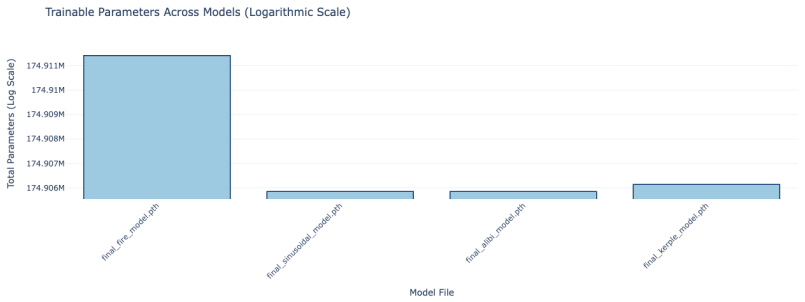


Figure 14: Comparison of trainable parameters across models (logarithmic scale).