
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

1 The purpose of this work is to develop a deeper understanding of GPT-2 by
2 building it from scratch and systematically analyzing techniques that improve its
3 performance. We pretrain a GPT-2 model built from scratch with 124 million
4 parameters on the 10-billion-token FineWeb-Edu dataset, leveraging distributed
5 Data Parallel (DDP) processing. To explore the impact of architectural choices,
6 we integrate a slew of positional encodings and attention mechanisms. Using the
7 HellaSwag benchmark, we assess model accuracy and scalability, while Weights
8 & Biases aids in analyzing attention maps, parameter distributions, and gradient
9 behavior. This work provides insights into performance differences and especially
10 dives deep into understanding what drives performance in GPT-like models.

11 **1 Introduction**

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

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

25 To optimize the training process, we utilized Distributed Data Parallel (DDP) processing across A5000
26 and A6000 GPUs, completing 20,000 epochs efficiently. Two advanced attention mechanisms, Flash
27 Attention and Linformer, were integrated to address the computational inefficiencies of traditional
28 self-attention while maintaining the model’s ability to capture long-range dependencies. Additionally,

29 six positional encoding schemes were evaluated: Baseline (Sinusoidal), Learned, RoPE, ALiBi, FIRE,
30 and Kerple to understand their role in encoding sequence order and improving generalization.
31 The performance of the model was benchmarked in HellaSwag, a challenging data set to test natural
32 language inference and reasoning. We picked HellaSwag because it shows the growth in performance
33 of the model at designated checkpoints in training which gives insights into scalability as well. To
34 gain deeper insights, we used Weights & Biases for analyzing attention maps, parameter distributions,
35 and gradient behaviors. Our findings highlight the influence of efficient attention mechanisms
36 and positional encodings on model performance, providing a foundation for future research into
37 optimizing large-scale language models.

38 2 Related Work

39 2.1 Transformer Architecture and GPT-2

40 The Transformer architecture, introduced by Vaswani et al. (2017), revolutionized natural language
41 processing by using attention mechanisms [9]. This architecture underpins large language models
42 like GPT-2, developed by OpenAI, which has 1.5 billion parameters and was trained on 8 million web
43 pages [7]. In this project, we implement the smallest version of GPT-2 due to resource constraints.
44 The core innovation in Transformers is the attention mechanism, particularly self-attention, which
45 allows the model to focus on relevant parts of the input sequence [9]. Multi-head attention further
46 enhances this capability by jointly attending to information from different representation subspaces.
47 Since Transformers lack inherent sequential processing, positional encodings (PEs) are used to
48 incorporate word order information. The original Transformer employed sinusoidal positional
49 encodings as a baseline [5]. Subsequent work introduced learned and relative positional encodings,
50 which often outperform sinusoidal PEs, especially for longer sequences [5].
51 As models have grown, researchers have focused on improving scalability through model parallelism,
52 efficient attention mechanisms like flash attention, and optimized training techniques [2, 9]. GPT-2
53 and similar models have excelled in tasks like text generation, summarization, and question answering,
54 evaluated using metrics such as perplexity and performance on downstream tasks [7].

55 3 Method

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

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

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

68 We adhered closely to the hyperparameters and training setup from the original GPT-2 paper. Specifi-
69 cally:

- 70 • **Optimizer:** AdamW with $\beta_1 = 0.9$, $\beta_2 = 0.95$, and $\epsilon = 10^{-8}$.
- 71 • **Learning Rate:** Cosine decay scheduling, starting at 6×10^{-4} and decreasing to 6×10^{-5} ,
72 with a linear warmup over 715 steps.
- 73 • **Weight Decay:** 0.1.
- 74 • **Gradient Clipping:** Maximum norm of 1.0.

- 75 • **Batch Size:** A total of 512K tokens, achieved through gradient accumulation across multiple
 76 steps, with a per-GPU batch size adjusted to fit available GPU memory.
 77 • **Precision:** Mixed-precision training using bfloat16 with TensorFloat-32 (TF32) enabled for
 78 matrix multiplications on compatible hardware.

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

83 **3.1 Positional Encodings**

84 **3.1.1 Learned Positional Embeddings**

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

89 **3.1.2 Sinusoidal Positional Encodings**

90 While not used in GPT-2, the original Transformer model introduced sinusoidal positional encodings
 91 [9]. These are fixed encodings defined by sine and cosine functions:

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

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

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

94 **3.1.3 RoPE**

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

$$R_{\Theta, \{q, k\}}(x_m, m) = \begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix} W_{\{q, k\}} x_m \quad (3)$$

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

103 **3.1.4 Attention with Linear Biases (ALiBi)**

104 ALiBi adds a penalty to the attention weight scores proportional to the distance between tokens
 105 [6]. The attention score between tokens i and j is calculated using the equation 4, where m is a
 106 head-specific scalar that is fixed during training.

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

107 **3.1.5 Functional Interpolation for Relative Positions (FIRE)**

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

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

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

$$f_\theta(x) = v_3^T \sigma(V_2 \sigma(v_1 x)) \quad (6)$$

113 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.
 114 Attention calculation incorporating FIRE is similar to ALiBi.

115 3.1.6 Kerple

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

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

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

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

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

123 3.2 Attention mechanisms

124 3.2.1 Regular Attention

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

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

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

131 3.2.2 Flash Attention

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

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

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

140 3.2.3 Efficient Attention

141 Efficient Attention [3] reduces the computational and memory complexity of self-attention from
 142 $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$, where N is the sequence length. This is achieved by expressing self-attention as a
 143 linear dot-product of kernel feature maps using $\phi(x) = \text{elu}(x) + 1$, ensuring positive similarity scores.
 144 The approach leverages the associativity property of matrix products to accelerate computation
 145 and reveals a connection to recurrent neural networks. However, this method primarily targets the
 146 encoder and does not directly address causal masking challenges in autoregressive decoders, which
 147 are essential for preventing information leakage from future tokens.

148 **3.2.4 Linformer**

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

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

154 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
 155 ($k \ll n$) and n is the sequence length.

156 **4 Results**

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

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

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

167 **4.1 Comparison Across Positional Encodings**

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



Figure 1: Best Training Loss

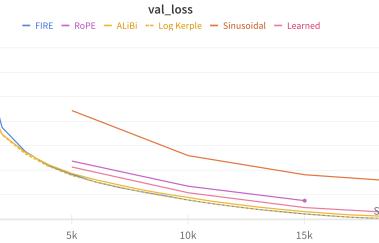


Figure 2: Validation Loss

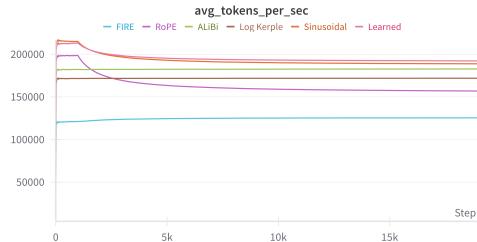


Figure 3: Average Tokens per Second

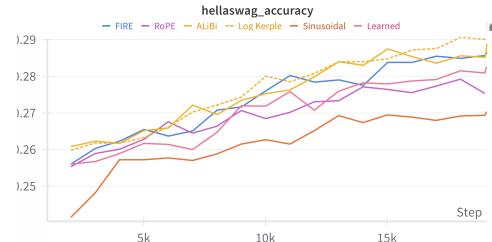


Figure 4: HellaSwag Accuracy

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

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

183 4.1.1 Discussion

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

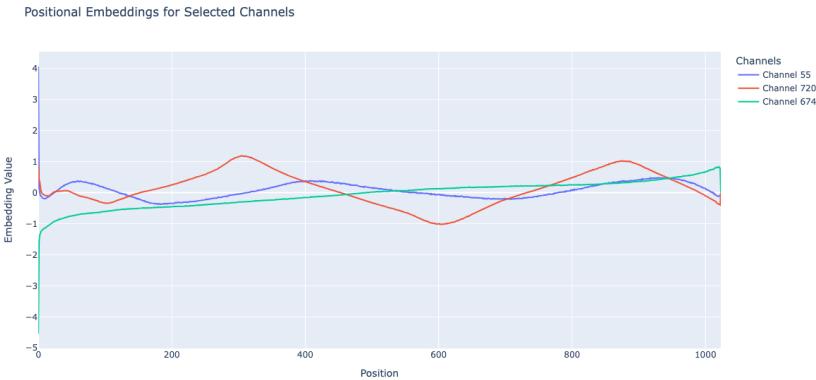


Figure 5: Visualization of selected channels from the learned positional embedding matrix, highlight-
ing sinusoidal-like patterns.

196 4.2 Comparison Across Attention Mechanisms

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

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

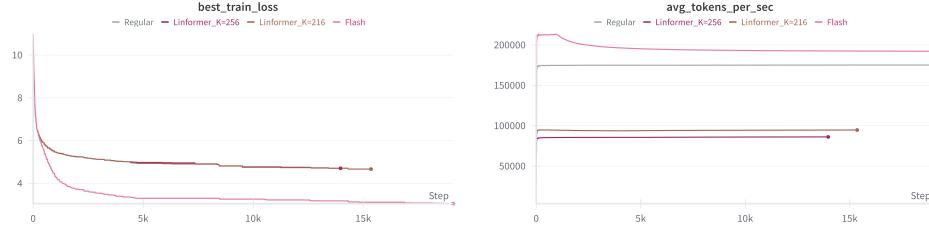


Figure 6: Training performance: Best train loss (left) and average tokens per second (right) across attention mechanisms.

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.

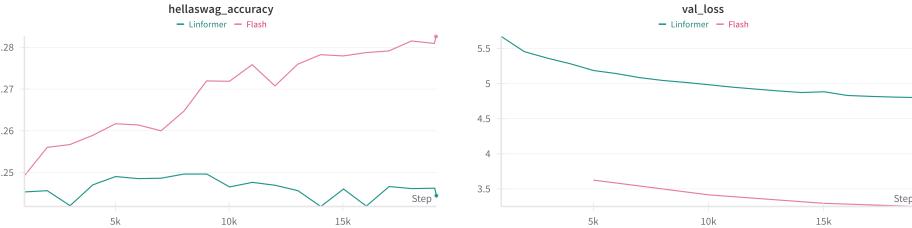


Figure 7: Testing performance: HellaSwag accuracy (left) and validation loss (right) across attention mechanisms.

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.

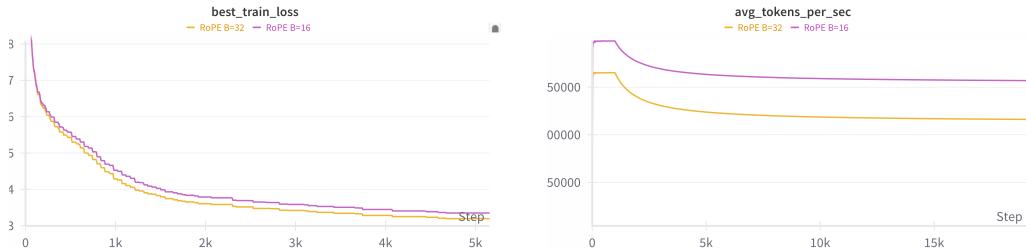


Figure 8: Training loss comparison for RoPE across $B = 16$ and $B = 32$.

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

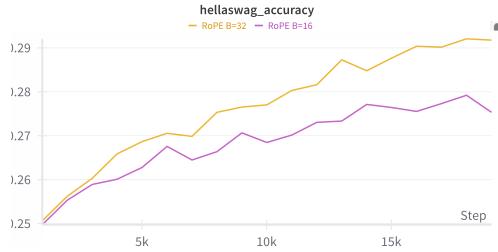


Figure 10: HellaSwag accuracy comparison for RoPE across $B = 16$ and $B = 32$.

224 The `avg_tokens_per_sec` metric (Figure 9) highlights the efficiency differences between batch sizes.
 225 $B = 32$ exhibits lower throughput compared to $B = 16$, which is expected due to the computational
 226 overhead associated with larger batches. The `hellawag_accuracy` plot (Figure 10) demonstrates that
 227 $B = 32$ consistently outperforms $B = 16$ during testing. This trend indicates that larger batch sizes
 228 may contribute to better generalization on downstream tasks, despite minimal differences in training
 229 loss.

230 **4.3.1 Discussion**

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

240 **5 Conclusion**

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

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

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

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

258 **References**

- 259 [1] Ta-Chung Chi, Ting-Han Fan, Peter J. Ramadge, and Alexander I. Rudnicky. Kerple: Kernelized
 260 relative positional embedding for length extrapolation, 2022.

- 261 [2] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and
 262 memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing*
 263 *Systems*, 35:16344–16359, 2022.
- 264 [3] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers
 265 are rnns: Fast autoregressive transformers with linear attention, 2020.
- 266 [4] Shanda Li, Chong You, Guru Guruganesh, Joshua Ainslie, Santiago Ontanon, Manzil Zaheer,
 267 Sumit Shanghai, Yiming Yang, Sanjiv Kumar, and Srinadh Bhojanapalli. Functional interpolation
 268 for relative positions improves long context transformers, 2024.
- 269 [5] Taro Miyazaki, Hideya Mino, and Hiroyuki Kaneko. Understanding how positional encodings
 270 work in transformer model. In *Proceedings of the 2024 Joint International Conference on*
 271 *Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages
 272 17011–17018, 2024.
- 273 [6] Ofir Press, Noah A. Smith, and Mike Lewis. Train short, test long: Attention with linear biases
 274 enables input length extrapolation, 2022.
- 275 [7] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al.
 276 Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.
- 277 [8] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer:
 278 Enhanced transformer with rotary position embedding, 2023.
- 279 [9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez,
 280 Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.
- 281 [10] Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention
 282 with linear complexity, 2020.

283 A Hardware and Resource Details

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

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

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

292 B Supplementary Results

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

298 We include the following:

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

303 B.1 Comparison Across Positional Encodings when B=32

304 In the results section earlier, we compared the performance of various positional encodings (PEs) for
 305 $B=16$. Here, we present the same comparison for $B=32$. However, FIRE is excluded due to its high

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

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

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

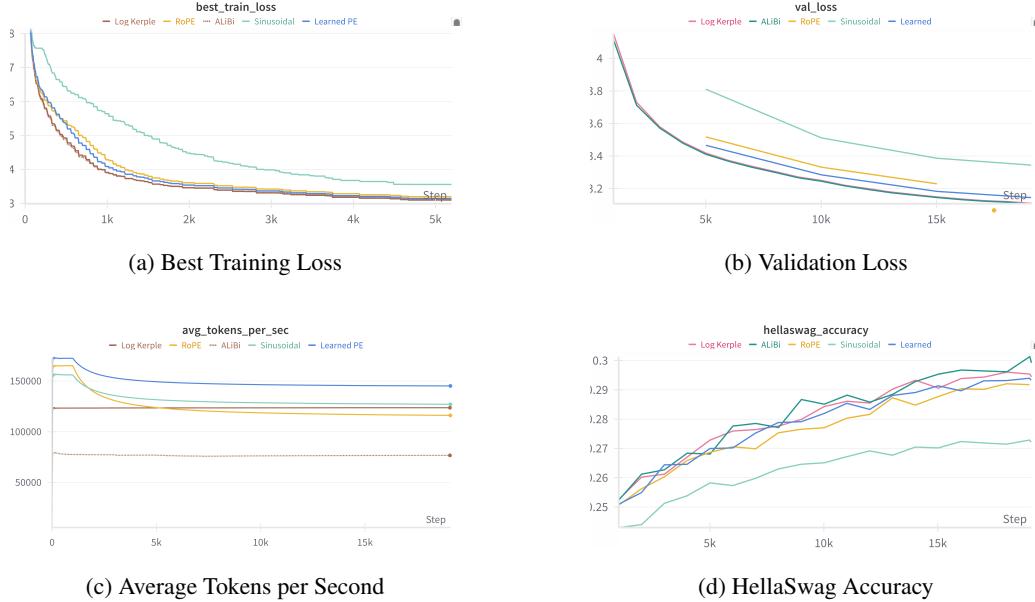


Figure 11: Performance comparison across positional encodings with $B = 32$.

314 B.2 Comparison Across Different Attention Mechanisms

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

320 B.2.1 Training Performance

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

325 B.2.2 Throughput Efficiency

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



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

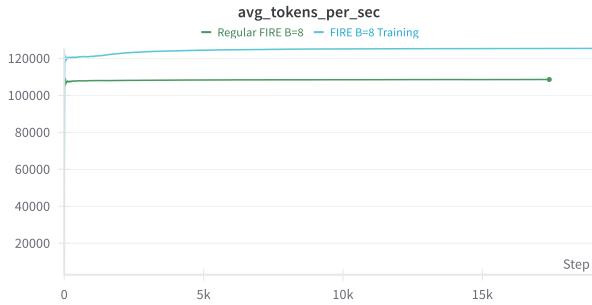


Figure 13: Throughput comparison between FIRE Regular and FIRE Flash with $B = 8$.

331 B.2.3 Key Observations

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

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

341 C Resource Limitations and Workarounds

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

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

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

356 **C.1 Comparison of Trainable Parameters Across Models**

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

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

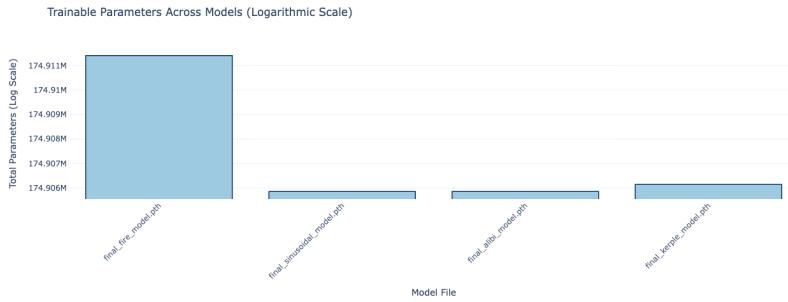


Figure 14: Comparison of trainable parameters across models (logarithmic scale).