

# Deep Learning Algorithms and Implementations Report

## Understanding and Improving GPT-2 Implementations in NanoGPT

Peiyou Liu<sup>1</sup>

<sup>1</sup> National Taiwan University,  
`t14902135@ntu.edu.tw`

### Abstract

This report examines the implementation of multi-head attention in NanoGPT and evaluates the effectiveness of KV caching for inference acceleration. It shows that NanoGPT efficiently computes Query, Key, and Value via a single linear projection, improving computational and memory efficiency. Experiments on Shakespeare and Chinese Tang poetry datasets—along with fine-tuning of GPT-2—demonstrate that KV caching yields significant speedups in large models by reducing redundant computation, while offering limited gains in small models. The memory and I/O overhead of KV caching scales linearly with sequence length but becomes negligible relative to computation savings in larger architectures.

Github Repository: [https://github.com/pyliu0311/NTU\\_CSIE7435\\_NanoGPTProject](https://github.com/pyliu0311/NTU_CSIE7435_NanoGPTProject)

## 1 Introduction

Transformers (Vaswani et al., 2017) have emerged as the cornerstone of modern deep learning, particularly in natural language processing (NLP), owing to their ability to capture long-range dependencies through self-attention mechanisms. The success of models like GPT (Radford et al., 2018), BERT (Devlin et al., 2019), and their successors underscores the importance of scalable attention architectures. Among their key innovations, multi-head attention (MHA)—introduced in the original Transformer paper—enables the model to jointly attend to information from different representation subspaces, enhancing both expressiveness and parallelizability. However, while the theoretical formulation of MHA is well-established (e.g., as a weighted sum of value vectors conditioned on query-key interactions), its practical implementation in large-scale systems often involves subtle but consequential design choices (Child et al. (2019), Dao et al. (2022)). Bridging this gap is critical for optimizing model performance, interpretability, and computational efficiency, especially in resource-constrained settings.

In this research, I investigate two interconnected dimensions of Transformer-based architectures, with a focus on NanoGPT<sup>1</sup>, a lightweight yet representative implementation of the GPT family. First, I rigorously analyze the alignment between the mathematical definition of multi-head attention and its practical instantiation in NanoGPT. By dissecting the codebase and comparing it against theoretical principles (e.g., query-key-value decomposition, scaling factors, and projection matrices), I identify implementation nuances that influence model behavior.

---

<sup>1</sup><https://github.com/karpathy/nanoGPT>

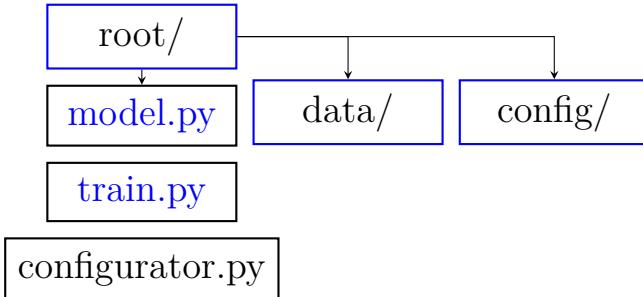


Figure 1: Core Code Structure Essentials of NanoGPT

Second, I address a pressing challenge in Transformer inference: computational redundancy during autoregressive generation. Specifically, I implement and evaluate KV (key-value) caching, a technique widely adopted in production systems (e.g., OpenAI’s GPT-3(Brown et al., 2020)). By profiling memory usage, latency, and throughput, I quantify the trade-offs between memory overhead and inference speedup, offering insights into efficiency optimizations for resource-constrained deployments.

This research contributes to the broader discourse on efficient Transformer systems by:

1. Grounding implementation choices in theory—clarifying how MHA’s mathematical operations translate to code.
2. Empirically validating KV caching—demonstrating its impact on inference efficiency while highlighting practical constraints.

Through this dual focus, I aim to advance the understanding of Transformer architectures not only as theoretical constructs but as engineered systems where algorithmic elegance meets real-world computational demands.

## 2 Multi-head Attention Implementation in NanoGPT

The mathematical formulation of multi-head attention in the original Transformer paper (Vaswani et al., 2017) is as follows:

$$\text{head}_i = \text{SoftMax}\left(\frac{ZW_Q^i(W_K^i)^\top Z^\top}{\sqrt{d}}\right)ZW_V^i \in \mathbb{R}^{T \times d/h} \quad (1)$$

$$\text{Output} = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W_O \in \mathbb{R}^{T \times d}$$

The multi-head attention mechanism processes an input sequence  $Z \in \mathbb{R}^{T \times d}$ , where  $T$  represents the sequence length and  $d$  denotes the embedding dimension. For each attention head  $\text{head}_i$ , the input  $Z$  is first projected into queries, keys, and values using learnable weight matrices  $W_Q^i, W_K^i \in \mathbb{R}^{d \times d/h}$  and  $W_V^i \in \mathbb{R}^{d \times d/h}$ , where  $h$  is the number of attention heads and  $d/h$  ensures dimension reduction per head. The scaled dot-product attention is computed as  $\text{SoftMax}\left(\frac{ZW_Q^i(W_K^i)^\top Z^\top}{\sqrt{d}}\right)$ , where the query-key dot product  $ZW_Q^i(W_K^i)^\top Z^\top$  yields a  $T \times T$  attention matrix, normalized by the scaling factor  $\sqrt{d}$  to stabilize gradients during training. The

resulting attention weights are then applied to the value projection  $ZW_V^i$  to produce the head output  $\text{head}_i \in \mathbb{R}^{T \times d/h}$ . Finally, the outputs of all  $h$  heads are concatenated and linearly transformed by  $W_O \in \mathbb{R}^{d \times d}$  to generate the final multi-head attention output  $\text{Output} \in \mathbb{R}^{T \times d}$ , preserving the original input dimension while capturing diverse attention patterns across heads.

In NanoGPT's implementation, the multi-head self-attention mechanism differs from the standard formulation described above. For clarity (while omitting the batch dimension), the following diagram concisely illustrates NanoGPT's approach to multi-head self-attention. In subsequent sections, we will focus on its mathematical formulation and analyze the advantages of this implementation.

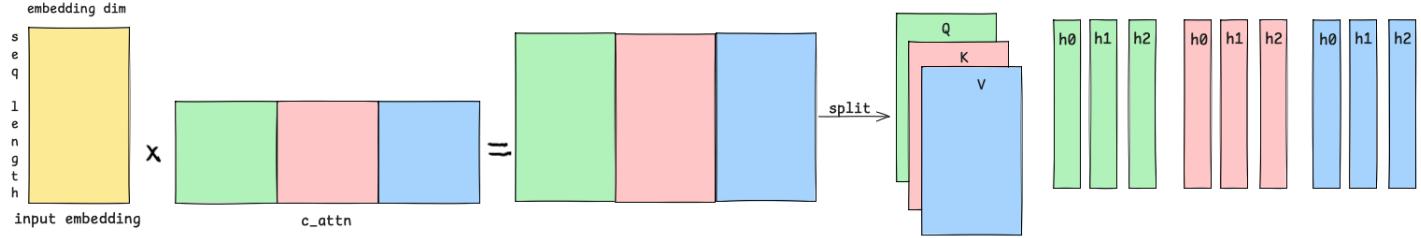


Figure 2: Illustration of Q, K, V generation and head split in NanoGPT

## 2.1 Computation of Query, Key, and Value in NanoGPT

In the implementation of NanoGPT, as evidenced by the referenced code segment, since the Query, Key, and Value in self-attention are all derived from linear transformations of the input  $Z$ , the authors directly allocate a consolidated weight matrix  $W_{attn} \in \mathbb{R}^{d \times 3d}$ . This enables the unified computation of  $Q$ ,  $K$ , and  $V \in \mathbb{R}^{T \times d}$  through a single matrix operation (i.e., a linear layer transformation).

```
self.c_attn = nn.Linear(
    config.n_embd, 3 * config.n_embd, bias=config.bias)
q, k, v = self.c_attn(x).split(self.n_embd, dim=2)
```

Figure 3: Implementation of Q, K, V Generation in NanoGPT

The mathematical formulation of this operation is as follows:

$$H = ZW_{attn} \in \mathbb{R}^{T \times 3d} \\ Q = H_{:,d}, \quad K = H_{:,d:2d}, \quad V = H_{:,2d:3d} \in \mathbb{R}^{T \times d} \quad (2)$$

It can be readily observed that, using the notation from (1), the above Query (Q), Key (K), and Value (V) can be equivalently expressed as:

$$W_{attn} = \text{Concat}(W_Q^1, \dots, W_Q^h, W_K^1, \dots, W_K^h, W_V^1, \dots, W_V^h) \in \mathbb{R}^{d \times 3d} \\ Q = H_{:,d} = \text{Concat}(ZW_Q^1, \dots, ZW_Q^h) \\ K = H_{:,d:2d} = \text{Concat}(ZW_K^1, \dots, ZW_K^h) \\ V = H_{:,2d:3d} = \text{Concat}(ZW_V^1, \dots, ZW_V^h) \quad (3)$$

The unified generation of Query (Q), Key (K), and Value (V) matrices through a single linear transformation, as implemented in NanoGPT, offers several computational and memory efficiency advantages over the per-head decomposition approach.

1. First, this method reduces computational overhead by consolidating three separate matrix multiplications (for Q, K, and V) into a single large-scale GEMM (General Matrix Multiply) operation. Modern hardware accelerators, such as GPUs, optimize such batched operations, minimizing kernel launch latency and leveraging parallelism more effectively. The unified approach thus improves throughput by avoiding the inefficiencies of multiple small matrix operations.
2. Second, memory efficiency is enhanced due to contiguous tensor allocation. By computing Q, K, and V as slices of a single output tensor, the strategy mitigates memory fragmentation and improves cache locality, particularly during attention score computation. Additionally, parameter storage remains compact, as the combined weight matrix  $W_{attn} \in \mathbb{R}^{d \times 3d}$  does not increase total memory usage compared to separate per-head weight matrices while simplifying memory access patterns.
3. Third, implementation complexity is reduced, as the unified approach eliminates the need for explicit per-head concatenation or reshaping operations. This leads to fewer potential sources of error and facilitates kernel fusion optimizations, where the linear transformation and tensor splitting can be executed within a single optimized GPU kernel, further reducing overhead.

## 2.2 Head Splitting and Tensor Shapes

Regarding Head Splitting and Tensor Shaping, the implementation in code operates as follows:

```
k = k.view(B, T, self.n_head, C // self.n_head).transpose(1, 2)
q = q.view(B, T, self.n_head, C // self.n_head).transpose(1, 2)
v = v.view(B, T, self.n_head, C // self.n_head).transpose(1, 2)
```

Figure 4: Head Splitting and Tensor Reshaping Implementation

Omitting the batch dimension, this operation is straightforward: it simply reshapes the tensors  $Q, K, V \in \mathbb{R}^{T \times d}$  into  $Q, K, V \in \mathbb{R}^{h \times T \times (d/h)}$ , where  $h$  denotes the number of attention heads.

Combining (1) and (3), this operation can be mathematically expressed using the stacking operator  $\text{Stack}(\cdot)$  along the first dimension as:

$$\begin{aligned} Q &= \text{Stack}(ZW_Q^1, \dots, ZW_Q^h) \in \mathbb{R}^{h \times T \times (d/h)} \\ K &= \text{Stack}(ZW_K^1, \dots, ZW_K^h) \in \mathbb{R}^{h \times T \times (d/h)} \\ V &= \text{Stack}(ZW_V^1, \dots, ZW_V^h) \in \mathbb{R}^{h \times T \times (d/h)} \end{aligned} \quad (4)$$

The reshape operation in multi-head attention enables parallel processing across attention heads by splitting the channel dimension ( $d$ ) into  $h$  subspaces (each of size  $d/h$ ). This allows each head to independently focus on different aspects of the input while maintaining computational efficiency through batched matrix operations.

The transpose operation serves a critical role in optimizing batched matrix multiplication by repositioning the multi-head attention dimension  $h$  to the first axis (transforming the tensor shape from  $[B, T, h, d/h]$  to  $[B, h, T, d/h]$ ). This adjustment ensures that subsequent batched matrix multiplications (e.g.,  $q @ k.\text{transpose}(-2, -1)$ ) operate efficiently on contiguous memory blocks. Such an optimized memory layout enhances cache utilization by maintaining sequential access along the  $T$  and  $d/h$  dimensions while aligning with GPU-parallelized computation patterns. Consequently, it minimizes memory stride overhead and significantly accelerates large-scale matrix multiplications in multi-head attention mechanisms.

### 3 Implement KV Caching in NanoGPT

KV cache (Key-Value cache) is an optimization technique employed in autoregressive transformer models to accelerate sequential token generation. During decoding, previous key-value pairs from the attention layers are cached to avoid redundant computation for past tokens. Commonly used in models like GPT (e.g., OpenAI’s GPT-3(Brown et al., 2020)), KV cache balances computational overhead and memory constraints during autoregressive decoding.

When a new token  $z_{T+1}$  is input, following (2) corresponding computational operations must be performed.

$$\begin{aligned} h_{T+1} &= z_{T+1}W_{\text{attn}} \\ q_{T+1} &= (h_{T+1})_{:d}, \quad k_{T+1} = (h_{T+1})_{d:2d}, \quad v_{T+1} = (h_{T+1})_{2d:3d} \end{aligned} \tag{5}$$

Then the Q, K, V matrices are updated as follows:

$$K_{T+1} = \begin{bmatrix} K_T \\ k_{T+1} \end{bmatrix} \in \mathbb{R}^{(T+1) \times d}, \quad V_{T+1} = \begin{bmatrix} V_T \\ v_{T+1} \end{bmatrix} \in \mathbb{R}^{(T+1) \times d} \tag{6}$$

where  $K_T$  and  $V_T$  are the cached key and value matrices up to time step  $T$ . Then the attention output for the new token is computed as:

$$\text{out}_{T+1} = \text{softmax} \left( \frac{q^{T+1} K_{T+1}^\top}{\sqrt{d}} \right) V_{T+1} \in \mathbb{R}^{1 \times d} \tag{7}$$

It is readily observable that during each forward propagation when computing new tokens, the previously cached Key-Value pairs ( $K_T, V_T$ ) can be systematically reused, thereby substantially reducing the computational overhead associated with linear projection operations required for generating K and V values.

In NanoGPT, when KV Cache is not employed, the model must recompute the entire forward pass over the current full sequence for every newly generated token. Specifically, given a sequence of length  $T$  that has been generated so far, the system recomputes the joint projection  $H = ZW_{\text{attn}} \in \mathbb{R}^{T \times 3d}$  to obtain the query (Q), key (K), and value (V) matrices for all  $T$  positions.

This projection incurs a time complexity of  $O(Td^2)$ . Subsequently, the attention computation—comprising the matrix multiplication  $QK^\top$  and the weighted aggregation with  $V$ —dominates with a complexity of  $O(T^2d)$ . Consequently, the per-step time complexity for generating the  $T$ -th token is  $O(Td^2 + T^2d)$ . When generating a full sequence of length  $L$ , summing over steps  $T = 1$  to  $L$  yields an overall time complexity of  $O(L^2d^2 + L^3d)$ , which exhibits cubic growth in  $L$  and severely hampers inference efficiency for long-sequence generation.

In contrast, under the KV Cache paradigm, the key and value matrices for previously generated tokens are computed once and persistently stored. Only the query vector  $q_T$  for the current new token undergoes a lightweight projection. In this regime, the projection at step  $T$  operates on a single token, reducing its cost to  $O(d^2)$ , while the attention computation simplifies to a dot-product between  $q_T$  and the cached  $K_{1:T} \in \mathbb{R}^{T \times d}$ , followed by a weighted sum over  $V_{1:T}$ , both costing  $O(Td)$ . Thus, the per-step time complexity is reduced to  $O(d^2 + Td)$ . Aggregating over a full sequence of length  $L$ , the total time complexity becomes  $\sum_{T=1}^L O(d^2 + Td) = O(Ld^2 + L^2d)$ , effectively lowering the dominant term from  $O(L^3)$  to  $O(L^2)$ . This optimization preserves the model’s representational capacity while substantially accelerating autoregressive inference.

While KV Cache facilitates efficient inference, it concurrently introduces substantial memory overhead that warrants careful consideration (Ge et al., 2023). In a multi-head attention architecture with  $N$  Transformer layers, each layer contains  $h$  attention heads, and the hidden dimension of the model is  $d$ . When processing a batch of  $B$  sequences, the KV Cache must maintain separate key and value matrices for each sequence in the batch. Assuming a maximum sequence length of  $L$  within the batch, each layer stores key and value tensors of shape  $B \times L \times d$ , corresponding to a total of  $2BLd$  floating-point elements per layer. Across all  $N$  layers, the cumulative memory footprint of the KV Cache becomes  $2NBLd$  elements.

This storage is in addition to the model parameters and scales linearly with batch size, sequence length, and model width. In large-batch or long-context inference scenarios, this overhead can dominate the total memory consumption, often exceeding the memory required to store the model weights themselves, thereby posing a critical constraint on deployable context lengths and throughput.

### 3.1 Training and Fine-tuning the NanoGPT Model

The experiment first trains a relatively small NanoGPT model on both the Shakespearean text dataset<sup>2</sup> provided by the NanoGPT authors and a Chinese Tang poetry dataset<sup>3</sup>, using the same training configuration for both corpora. To further investigate whether the performance improvement from KV caching becomes more pronounced in larger models—since the memory overhead of the cache becomes relatively smaller compared to the total number of model parameters—the study proceeds to fine-tune a GPT-2 model on the Shakespeare dataset.

Detailed experimental configurations, including base model settings, training hyperparameters, and fine-tuning parameter specifications, are presented as follows.

---

<sup>2</sup><https://github.com/karpathy/nanoGPT/blob/master/data/shakespeare/prepare.py>

<sup>3</sup><https://github.com/chinese-poetry/chinese-poetry>

Table 1: Experimental Setup Configuration

Category	Specification
GPU	<ul style="list-style-type: none"> <li>Model: NVIDIA GeForce RTX 5080 (WDDM driver)</li> <li>Driver Version: 572.61</li> <li>CUDA Version: 12.8</li> <li>VRAM: 16,303 MiB</li> </ul>
Operating System	Windows 11
Software	Python 3.9

Table 2: Training Configuration

Parameter	Description	Value
eval_interval	Evaluation frequency (iterations)	25
batch_size	Samples per batch	64
block_size	Context window length (tokens)	1024
n_layer	Number of transformer layers	6
n_head	Number of attention heads	6
n_embd	Embedding dimension size	384
dropout	Dropout probability	0.2
learning_rate	Optimization step size	$1 \times 10^{-3}$
max_iters	Total training iterations	5000

Table 3: Fine-Tuning Configuration

Parameter	Description	Value
eval_interval	Evaluation frequency (iterations)	1
init_from	Pretrained base model	gpt2-xl
batch_size	Samples per forward pass	1
gradient_accumulation_steps	Effective batch size multiplier	32
max_iters	Total training iterations	20
learning_rate	Optimization step size	$3 \times 10^{-5}$

- Effective batch size =  $\text{batch\_size} \times \text{gradient\_accumulation\_steps} = 32$
- Using gpt2-xl (1.5B parameter version) as foundation model
- Frequent evaluation (every iteration) for monitoring fine-tuning progress

Below presents the variation curves of the loss function during both the model training process and validation phase.

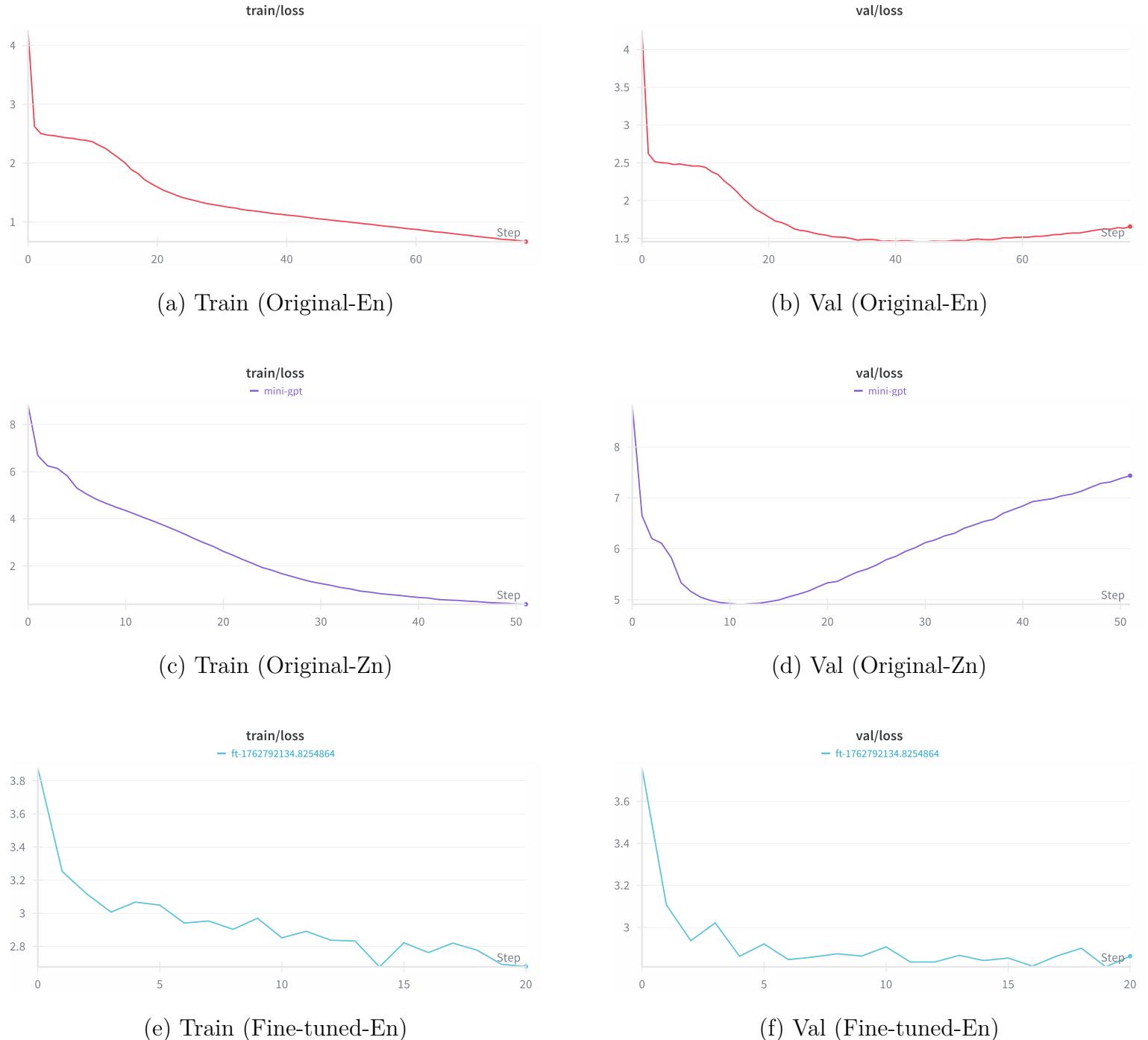


Figure 5: Training and validation loss curves

Noting the overfitting indicated by the validation loss in Figure 5d, the checkpoint corresponding to the lowest validation loss was selected as the final trained model.

### 3.2 Experiment

This study conducts controlled experiments by varying the max\_token parameter across multiple test samples, culminating in the computation of essential statistical metrics. Specifically, in each of the three aforementioned scenarios, the total inference time, QKV generation time, attention

computation time, KV cache write time (cache mode only), and KV cache GPU memory usage (cache mode only) were evaluated with and without caching.

### (i) NanoGPT on Shakespeare Dataset

The model uses the configuration specified in Table 2, with a dataset size of approximately 1.0 million tokens for training and approximately 0.11 million tokens for validation.

Table 4: Performance Metrics for Token Generation (NanoGPT on Shakespeare Dataset)

max_tokens	cache	total_time	qkv_time	attn_time	tokens_per_sec	num_samples
1000	False	4.11	0.35	1.13	243.14	10
1000	True	3.11	0.28	0.83	321.07	10
2000	False	12.08	1.11	3.41	165.53	10
2000	True	6.12	0.54	1.60	326.79	10
4000	False	21.45	1.35	4.17	186.45	10
4000	True	12.26	1.07	3.23	326.30	10
8000	False	42.23	2.07	6.36	189.45	10
8000	True	24.70	1.85	5.53	323.93	10

Table 5: KV Cache Cost (NanoGPT on Shakespeare Dataset)

max_tokens	cache_update_time	memory_usage
1000	0.25	17.58MiB
2000	0.53	35.16MiB
4000	1.12	70.32MiB
8000	1.89	140.64MiB

### (ii) NanoGPT on Tang Poetry Dataset

The model uses the configuration specified in Table 2, with a dataset size of approximately 0.4 million tokens for training and approximately 0.04 million tokens for validation. However, due to the nature of the Chinese dataset, it contains a larger number of unique tokens.

Table 6: Performance Metrics for Token Generation (NanoGPT on Tang Poetry Dataset)

max_tokens	cache	total_time	qkv_time	attn_time	tokens_per_sec	num_samples
1000	False	5.10	0.45	1.43	196.22	10
1000	True	3.07	0.27	0.81	325.70	10
2000	False	12.15	1.11	3.42	164.55	10
2000	True	6.10	0.54	1.61	327.70	10
4000	False	21.45	1.25	3.87	186.47	10
4000	True	12.24	1.07	3.24	326.80	10
8000	False	41.73	2.07	6.41	191.71	10
8000	True	24.75	1.75	5.85	323.23	10

Table 7: KV Cache Cost (NanoGPT on Tang Poetry Dataset)

max_tokens	cache_update_time	memory_usage
1000	0.24	17.58MiB
2000	0.49	35.16MiB
4000	1.06	70.32MiB
8000	1.68	140.64MiB

### (iii) GPT-2 on Shakespeare Dataset

The model uses the configuration specified in Table 3, with a dataset size of approximately 1.0 million tokens for training and approximately 0.11 million tokens for validation.

Due to excessively long runtime, only experiments with a smaller maximum number of tokens were tested here; however, the results are already sufficiently clear.

Table 8: Performance Metrics for Token Generation (GPT2 on Shakespeare Dataset)

max_tokens	cache	total_time	qkv_time	attn_time	tokens_per_sec	num_samples
10	False	5.91	0.02	0.34	1.69	10
10	True	3.57	0.15	0.16	2.80	10
20	False	14.79	0.04	0.39	1.35	10
20	True	7.14	0.08	0.53	2.80	10
40	False	42.22	0.87	2.07	0.95	10
40	True	14.34	0.39	1.13	2.79	10
80	False	108.77	3.44	15.64	0.74	10
80	True	28.83	0.38	2.63	2.77	10
160	False	261.91	6.15	31.02	0.61	10
160	True	69.70	1.45	3.81	2.30	10

Table 9: KV Cache Cost (GPT2 on Shakespeare Dataset)

max_tokens	cache_update_time	memory_usage
10	0.08	5.91MiB
20	0.17	11.82MiB
40	0.32	23.64MiB
80	0.68	47.28MiB
160	1.29	94.56MiB

### 3.3 Analysis

Based on the experimental results above, I present the following analysis.

1. Based on the results in Tables 4 and 6, for relatively smaller models, KV Cache yields modest improvements in the two key components that typically benefit from acceleration—KV generation and attention computation—though the magnitude of improvement is limited. The primary source of speedup appears to stem from the reduction in concatenation operations due to incremental updates. It is hypothesized that, given the small size of Model D, the theoretical benefits of KV Cache are not fully realized, possibly due to PyTorch’s optimizations for batched matrix operations and the GPU’s computational capacity, which already enable efficient execution even without caching.

Additionally, I conducted partial experiments with the smaller model on CPU, and the results showed a more pronounced speedup from KV Cache, which partially corroborates my hypothesis.

2. As shown in Table 8, when  $d$  is large, the computational complexity shifts significantly: the cost changes from  $O(L^2d^2 + L^3d)$  without KV Cache to  $O(Ld^2 + L^2d)$  with KV Cache—an improvement that becomes substantial. Moreover, the larger number of layers in the model further amplifies this effect. Consequently, for this larger model, KV Cache demonstrates clear advantages in both KV generation and attention computation, leading to significant inference acceleration.

Admittedly, the small maximum token count results in relatively little additional overhead from KV cache management operations. However, this overhead is negligible compared to the total inference time without caching in this setting and can thus be disregarded.

3. As shown in Tables 5, 7, and 9, the additional I/O time introduced by KV cache is nearly linearly proportional to its memory footprint. When the model is small (i.e., small  $d$ ), this overhead can be relatively significant; however, when the model is large (i.e., large  $d$ ), the relative impact of this I/O cost diminishes compared to the dominant computation time, rendering it comparatively negligible.

## 4 Conclusion

This work bridges the gap between theoretical formulations and practical implementations of Transformer-based language models, with a dual focus on architectural design and inference optimization. First, it clarifies how NanoGPT’s streamlined implementation of multi-head attention—using a single linear layer to jointly produce Query, Key, and Value—faithfully aligns with the original Transformer mathematics while offering tangible benefits in computational efficiency, memory layout, and code simplicity. Second, it demonstrates that KV caching delivers substantial inference speedups in large models (e.g., GPT-2-XL) by eliminating redundant re-computation, though its impact is marginal in small models due to hardware and framework-level optimizations. Together, these insights underscore a key principle: effective deployment of deep learning models requires not only algorithmic understanding but also careful consideration of implementation-level choices. The findings provide practical guidance for researchers and engineers aiming to balance performance, memory, and code clarity when working with or extending minimal yet representative GPT implementations like NanoGPT.

## References

- Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.
- Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.
- 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.
- Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186, 2019.
- Suyu Ge, Yunan Zhang, Liyuan Liu, Minjia Zhang, Jiawei Han, and Jianfeng Gao. Model tells you what to discard: Adaptive kv cache compression for llms. *arXiv preprint arXiv:2310.01801*, 2023.
- Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.
- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.