KV-ALIGN: RECTIFYING ABSOLUTE POSITION DATA IN KV-CACHES FOR STREAMING LLMS

Paul-Andrei Aldea Michael Hwang John Kim Jason Liang

https://github.com/eecs585projectorg/kv-align

ABSTRACT

The outputs of large language models (LLMs) are generally constrained by the maximum context length for which the model was trained. Significantly extending this limit is challenging, typically requiring partial model retraining (Wang et al., 2024). One alternative to generate unbounded, sensible outputs is StreamingLLM (Xiao et al., 2024), which uses a combination of attention sink and sliding window approaches while also providing a partial solution to dealing with outdated positional information in the KV-cache for RoPE and ALiBi embeddings. Although StreamingLLM's success is remarkable, it is not generalizable to every model (such as DistilGPT2) and has room for improvement in output readability (tomaarsen, 2023a;b). We introduce KV-Align, a new approach to the streaming problem that leverages a small neural network to realign positional information in the KV-cache entries upon token eviction during streaming. KV-Align is, to our knowledge, the first work capable of adjusting the KV-cache entries independently of the positional embeddings utilized by a model. For DistilGPT2, KV-Align improves perplexity by up to 10.74x over StreamingLLM, and achieves a 2.7x speed-up over a full attention recomputation. Even for a RoPE-based model, such as SmolLM2, we observe a significant readability improvement in its output. As such, KV-Align offers a new scalable solution for extending LLM output lengths.

1 INTRODUCTION

Large Language Models (LLMs) are artificial intelligence systems capable of processing and generating human-like texts, which have become increasingly prominent in the systems research field and in everyday applications by AI-users (Bubeck et al., 2023). In general, traditional LLM outputs are bounded by a context length — a maximum token-wise length for which the model was trained. A recent approach given by StreamingLLM (Xiao et al., 2024) has sought to increase the lengths of LLM outputs and have been largely successful in achieving unbounded text generations. To enable *streaming*[1] behavior, StreamingLLM leverages two key observations (for more details, see section 2.3.3):

- LLMs place a large share of attention on the beginning tokens of an input sequence (Xiao et al., 2024). Such tokens are called *attention sinks*.
- Relative position embeddings of tokens must be re-aligned during streaming if any evictions from the KV-cache occurs.

In addition, LLMs commonly leverage a KV-cache capable of storing block-wise and head-wise intermittent results, as described in section 2.1. Since the KV-cache is of finite size, infinite token generation requires a careful handling of cache data. In particular, StreamingLLM utilizes a sliding window approach as described in section 2.3 to achieve streaming behavior. However, StreamingLLM's implementation does not clearly generalize to all LLMs (in particular, to those with absolute position embeddings like DistilGPT2 (distilbert, 2024)), and even for its target models (like RoPE-based models discussed in section 2.2.2), there is a room for improvement in the readability of the model's output (tomaarsen, 2023a;b).

We introduce KV-Align, a novel approach to adjusting the KV-cache entries regardless of the type of positional embedding. To our best knowledge, this is the first work which achieves unbounded output generation for models with absolute positional embeddings (section 2.2.1) with significantly lower perplexity than StreamingLLM (up to a factor of 10.74) while being faster than the attention recomputation approach (up to a factor of 2.70). Namely, we achieve infinite token generation with DistilGPT2 (distilbert, 2024). KV-Align leverages a small neural network, detailed in section 3, to rectify both keys and values within the KV-cache which contain incorrect positional data. We then show that our approach is also applicable to other types of positional embeddings by evaluating performance on SmolLM2 (Allal et al., 2024), a RoPE-based model. Additionally, we experimentally observed that KV-Align takes very little time

[1] *Streaming* traditionally refers to the ability of a model to incrementally deliver tokens as they are being generated. For this work, we expand this definition to include the production of *unbounded* data.

(e.g., around 1–6 hours) and resources to train, and does not require thorough hyperparameter tuning. Additionally, KV-Align does not condition on the prompt-type and performs well even on dataset very different from the training data (section 5.3).

We start by presenting related works and motivations in section 2. Then, we explain details of KV-Align in sections 3 and 4. We present a variety of evaluations in section 5 and discuss future directions in section 6.

2 Background and Motivation

The key motivating problem arises from the interaction between position embeddings and the KV-cache of a large language model at the time of token eviction during streaming. Since the problem we deal with is quite technical in nature, we begin with a necessary overview of the relevant terms.

2.1 LLM and KV-cache

We take a highly simplified view of a transformer-based large language model that should nonetheless provide sufficient knowledge for the ensuing presentation[2]. For our purposes, the input to an LLM is a sequence of vectors $w_1, w_2, \ldots, w_n$, where we may imagine each vector represents an English word via some token embedding matrix that specifies the conversion of each English word to a vector. In particular, $w_1, \ldots, w_n$ can be thought of as the input prompt to the LLM, which may be an instruction or question. These vectors are then processed through a sequence of B attention blocks, each of which takes in a sequence of n vectors and outputs another sequence of n vectors (for DistilGPT2 model, $B = 6$). The final output of the LLM is roughly given as follows: take the final vector in the output of the last attention block, and feed it through a projection layer to convert it to vector w'_n, whose dimension should equal the number of tokens in our vocabulary. Essentially, w'_n is now used to generate the "next token" w_{n+1} that follows the initial sequence $w_1, \ldots, w_n$. There are multiple ways of generating w_{n+1} from w'_n. The greedy approach is to take the argmax of w'_n as the index of the next token, but a more sensible way is to perform multinomial sampling based on the top-K largest entries of w'_n after applying the softmax function (Hugging Face, 2024a; Karpathy, 2024). The whole procedure is then repeated (in the second step, for example, the input to the LLM is $w_1, \ldots, w_n, w_{n+1}$), producing a stream of words that, for a good LLM, should be readable and informative.

We now elaborate further on the inner workings of an attention block and introduce the KV-cache. Within each block is a collection of H attention heads (for DistilGPT2, $H = 12$), each of which admits as input a sequence of n vectors. Consider any head with input $u_1, \ldots, u_n$. The attention head performs a computation that converts each u_i's to three vectors: key k_i, value v_i, and query q_i. We define the **KV-dimension** d to be the dimensions of keys and values vectors[3] (for DistilGPT2, $d = 64$). These keys, values, and queries are used to compute the output sequence of the head. During the process, which may be termed the *attention computation*, the dot products are taken between each query vector q_i and the preceding key vectors $k_1, \ldots, k_i$, after which we apply the softmax function to the resulting vector of dot products to normalize its sum to 1; informally, this computes to what degree query i "pays attention" to the preceding keys. Subsequently, linear combinations of value vectors are taken with the attention scores as coefficients. The exact details of this procedure are not important for this paper, but one fact is pertinent: the attention mechanism functions in such a way that the computation of w'_n (and thus w_{n+1}) can be done without recomputing $q_1, \ldots, q_{n-1}$ (i.e., among the query vectors, only the last one needs to be computed), as long as we have cached $k_1, \ldots, k_{n-1}, v_1, \ldots, v_{n-1}$ from the previous steps of the generation. Such a cache is named a **KV-cache**. Most importantly, keeping a KV-cache allows us to compute w_{n+1} by providing only w_n as the input to the LLM, a fact we take advantage of throughout this paper.[4]

2.2 Position Embedding

Given an input sequence of tokens $w_1, w_2, \ldots, w_n$, it is intuitive that the positional information of the tokens may carry important linguistic information (e.g., w_1 precedes w_2, and here perhaps w_1 is an adjective modifying the noun w_2). Thus, large language models have a built-in way to insert such positional information to one or more parts of the model's computational pipeline. We focus on two different approaches for this: absolute position embedding and rotary position embedding (RoPE) (Su et al., 2024), where the latter is an example of a more general approach known as relative position embedding. An excellent reference for this subsection is a Stanford lecture given by Christopher Potts (Stanford Online, 2023).

2.2.1 Absolute Position Embedding

The absolute position embedding approach simply keeps track of a position embedding matrix with C rows, where row i contains a vector p_i that has the same dimension as

[2] One notable reference for this subsection is a video and example code by Karpathy that carefully reconstructs GPT2 (2024; 2024)

[3] Note that the query vector may have a different dimension than that of the key vector due to mechanisms like grouped-query attention (Ainslie et al., 2023).

[4] An explicit example of such an implementation, along with a discussion of issues to be wary of when using the KV-cache, can be found in (gante, 2023).

the initial token embedding vectors. Here C is called the **context length**, the upper bound on the length of the LLM's input and output sequences (unless we use some clever streaming techniques described in section 2.3); for DistilGPT2, $C = 1024$. Thus, rather than using $w_1, \ldots, w_n$ (i.e., a sequence of token embedding vectors) as the input to the LLM as before, we simply use $w_1 + p_1, \ldots, w_n + p_n$ as the input (Karpathy, 2024). Several models, such as GPT2 (Radford et al., 2019), and newer larger models like OPT (Zhang et al., 2022), use absolute position embeddings (Sinha et al., 2022). In this paper, we work with a smaller model named DistilGPT2 with 124 million parameters (distilbert, 2024; Sanh et al., 2019).

2.2.2 *RoPE*

An alternative approach known as RoPE (Su et al., 2024) adds positional information not at the very start of the input to the transformer, but rather at every head of every block. Simply stated, rather than taking the dot product of a key k_i and a query q_j (where the subscripts indicate the position of such vectors in the relevant sequence), we compute $k_i^T R_{i-j} q_j$, where R_{i-j} is an orthogonal matrix. Importantly, the entries of R_{i-j} only depend on the relative position difference $i-j$ between key and query indices, so RoPE is an example of a family of approaches known as relative position embedding. In our paper, we use SmolLM2-135M (Allal et al., 2024) as the primary representative example.

2.3 StreamingLLM

Recall that out of the box, an LLM's output is upper bounded by the context length C. The authors of (Xiao et al., 2024) propose a method called *StreamingLLM* for generation with an unbounded output length. Implementations of their approach can be found in (mit-han-lab, 2024; tomaarsen, 2023a).[5] Before listing some key contributions of StreamingLLM, we first clarify the goal of infinite streaming.

2.3.1 *Clarification of the Streaming Objective*

As in (Xiao et al., 2024; mit-han-lab, 2024), we do not aim to build a streaming model that can remember all of its previous tokens, contexts, and initial prompt throughout the generation. Rather, we simply aim to achieve streaming results in which the model's outputs do not often degenerate into unreadable sentences. For example, Figure 1 illustrates a type of output which we would like to avoid.

[5]The former is the authors' implementation; the latter is by the Github user tomaarsen.

Figure 1. Example of a Bad Streaming Output

```
...As the editor's history of The
Second, Bill B. The author, But.  And,
M. It's In a "C-B-In-I-C: S-L. "T (The
'L " In-A-S C-T." (E-E. B-2." - "A:
(H. S. A. (a.  H. T. E-1.  1, B2, the
original) "R.1-4 - A- C6) 'G "2 "L,
"E1") "M1" ).  One of its own, and its
name (M3.  L" (3." "H-9-
```

2.3.2 *Attention Recomputation*

One elementary way to perform infinite streaming is to discard all previous computations and redo attention computations upon every token eviction. To elaborate, fix some window size $M < C$, also called the **cache size** following (mit-han-lab, 2024). Once our window overflows, i.e., we have so far generated the tokens $w_1, \ldots, w_M, w_{M+1}$ including those for the input prompt, then we evict w_1 from the window. Now we consider our input sequence to be $w_2, \ldots, w_{M+1}$, and feed this sequence through the LLM to produce w_{M+2}. Then, we evict w_2 and the process repeats. Although this approach will guarantee a highly readable output, it does not make sense from the computational perspective due to the quadratic time complexity of attention calculations (coming from the $O(n^2)$-many dot products $k_i \cdot q_j$). Note that the attention recomputation method does not make use of the KV-cache at all.

2.3.3 *StreamingLLM's Contributions: Attention Sink and Realignment of Positional Information for RoPE and ALiBi*

StreamingLLM (Xiao et al., 2024) proposes a method that has linear time complexity while keeping the output highly readable. To understand StreamingLLM's contributions, we must first introduce a basic sliding window strategy and understand the issues with it. This basic method proceeds exactly as for the attention recomputation, but we save the KV-cache throughout the process. Let $M < C$ be the cache size as before. Once we have produced w_{M+1} (i.e., once the window overflows), the KV-cache will have stored keys and values for the first M indices (for each head in each attention block of the transformer). Then, we can evict the keys and values with the index one in the KV-cache, and then to generate w_{M+2}, we can simply pass to the model the revised, shortened KV-cache along with w_{M+1}, effectively "pretending" that the KV-cache stores information about first M tokens and we are about to generate $(M+1)^{\text{st}}$ token (even though we are really generating $(M+2)^{\text{nd}}$ token). The process may then inductively repeat forever. It is deceptively easy to incorrectly reason that this approach will generate an output which is theoretically equivalent to

the attention recomputation approach. However, the sliding window approach fails spectacularly in practice due to two subtle reasons (Xiao et al., 2024):

- First, suppose that the dot product $k_1 \cdot q_{M+1}$ is much larger than any of $k_2 \cdot q_{M+1}, \ldots, k_M \cdot q_{M+1}$. Then, evicting k_1 from the KV-cache and only taking the softmax of $(k_2 \cdot q_{M+1}, \ldots, k_M \cdot q_{M+1})$ can result in a nonsensical output. After all, the remaining keys in the KV-cache are autoregressively produced in a way that any attention computation with them likely makes sense only if we include k_1 in the computation, and making matters worse, by our assumption, k_1 most heavily influenced the original attention calculation! Note that we would not have such a problem if we evicted w_1 at the token-level (as opposed to evicting index-1 keys and values at the KV-cache-level) and performed attention recomputation with tokens $w_2, \ldots, w_M, w_{M+1}$ afresh, as this would recompute all keys and values without any lingering influence of w_1.
- Secondly, and more importantly for this paper, if we wish to emulate the attention recomputation method, then we must carefully re-insert the revised positional information. For example, in attention recomputation, after evicting the first token from $w_1, w_2, \ldots, w_{M+1}$, now w_2 is really the *first* token in the resulting list. Similarly, when we evict from the KV-cache, we encounter a problem that the cached keys and values are based on the positional information that are now outdated. This necessitates some careful manual fix.

StreamingLLM attempts to solve the first problem by utilizing the observation that for many attention heads, the attention scores for query with index $i \geq 5$ and the keys with indices $1-4$ are significantly close to 1, a phenomenon they dub *attention sink*. As a result, StreamingLLM perpetually maintains the keys/values indexed $1-4$ in the KV-cache, only evicting the subsequent tokens. Experimentally, this is verified to produce, on average, the necessary evictions from the KV-cache which do not noticeably damage the output quality (Xiao et al., 2024).

Moreover, StreamingLLM provides a partial solution to the second problem for positional embeddings like RoPE and ALiBi. For example, for RoPE, the authors cache k_i before any orthogonal matrix is applied in the attention computation, where we ask the reader to recall the modified dot product formula $k_i^T R_{i-j} q_j$ for RoPE from section 2.2.2. After we evict the fifth token from the cache, the StreamingLLM approach would now use the cached k_i as is, but then compute $k_i^T R_{i-1-j} q_j$ instead. Here the key difference is the change in the relative index in the orthogonal matrix, reflecting the fact that after token eviction, k_i has now shifted to being the $(i-1)^{\text{st}}$ token in the sequence (Xiao et al., 2024).

2.4 Our Objective: A Position-Embedding-Agnostic Approach to Realign the Outdated KV-cache

The ideas behind the above two fixes are remarkable discoveries, and the authors of StreamingLLM note that the resulting model performs well with respect to the standard perplexity benchmark as well as the author's StreamEval benchmark — indicating a good streaming quality (Xiao et al., 2024). However, there are two key issues that are unresolved by StreamingLLM, which motivates KV-Align:

- The applicability of StreamingLLM depends on the exact implementation of the position embedding. For example, one reason StreamingLLM functions well for RoPE-based models is that there is a clear way to insert the "revised" positional information into every attention head after token eviction (i.e., by appropriately changing the orthogonal matrix). However, for absolute position-based models—where the positional information is only inserted at the input stage—there is no clear way to realign keys and values with the revised positions after token eviction, as observed by a Github issue (tomaarsen, 2023b). In fact, the subpar output in Figure 1 is generated with the StreamingLLM approach for DistilGPT2 (where we only adapted the attention sink idea but not the positional revision idea of StreamingLLM, as the latter is simply non-existent for absolute position embedding)[6]. Recalling the notation $w_i + p_i$ from section 2.2.1, note that if k_i is just a linear function of $w_i + p_i$, i.e., $k_i = A(w_i + p_i)$, then the solution to this problem for keys in the KV-cache is trivial: We just have to update $k_i \leftarrow k_i - Ap_i + Ap_{i-1}$ to subtract off the original positional information Ap_i and then add back the revised positional information Ap_{i-1}. However, especially for later blocks of the LLM, the keys and values are nonconvex functions of the input's token and position embeddings, so we are presented with a much more difficult problem.
- Even for RoPE embedding, the StreamingLLM approach of fixing the positional information is imperfect. This is for the simple reason that following a token eviction, the cached keys from later attention blocks are still based on the keys/values in the previous blocks

[6]We note that even with our newer idea of KV-Align, we encounter difficulties in consistently generating highly readable outputs using DistilGPT2, possibly due to the small size and older architecture of the model. However, we later demonstrate that KV-Align enjoys significantly smaller perplexity than StreamingLLM for DistilGPT2, which marks an important progress in the future direction of readability. We also present a readability improvement result for SmolLM2, a RoPE-based model.

with outdated positional information. Therefore, we may hope that with a more accurate method of fixing the KV-cache, we can generate a more readable output than StreamingLLM. We present a positive result in this direction in section 5.

This sets up the motivations for developing a method of realigning the outdated KV-cache during token evictions throughout streaming, in such a way that the method is completely independent of the position embedding methodology. The result is our method, KV-Align, which applies such a fix using a neural network, as further discussed in sections 3 and 4. An intuitive observation is that for such a network to be practical, it cannot be too small or too large: In the former case, the realignment will have poor accuracy, and in the latter case, we may as well perform the attention recomputation. It is also unclear how much conditioning on the inputs is required. Although distinct from our work, we draw some inspiration from a previous work known as Deja Vu (Liu et al., 2023) which showed that a small network can be trained to forecast certain behaviors of attention heads, illustrating that the highly complex nature of the attention mechanism can be learned by a simpler system. Our driving motivation is that a similar methodology may work in realigning positional information.

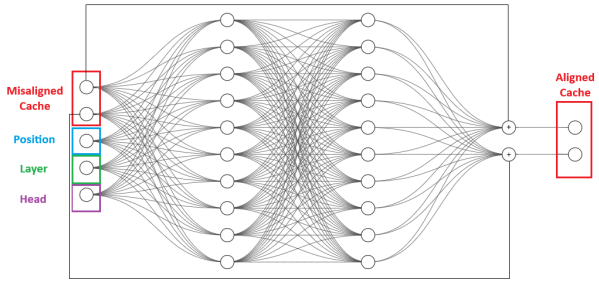
3 Solution Overview

KV-Align uses a neural network to realign the keys and values within the KV-cache after each token eviction, which occurs whenever we exceed the cache size of the sliding window approach (see section 2.3.3). Since keys and values serve fundamentally distinct roles in the attention computation, we trained two separate models, one for aligning keys and the other for aligning values. We refer to them as the **key network** and **value network**, respectively. The architecture of our model is relatively limited, given the latency-critical nature of KV-cache adjustments; in particular, we need to outperform attention recomputation (section 2.3.2). As such, the architecture is a feed-forward network with just two hidden layers. The inputs to the model consist of the misaligned cache entry, the position of the entry to be adjusted, along with the indices of the head and layer corresponding to the key or value. The output of the model is the re-aligned cache entry. A representation of our model, along with the inputs and outputs, can be seen in Figure 2. We provide further details on loss functions, optimizers, and hyperparameter tuning results in section 4.

Importantly, our architecture incorporates a residual connection from the initially misaligned keys (or values) to the final output[7]. That is, if f denotes the two-layer neural network and x, p, b, h denote the misaligned key (or value), position index, block index, and head index, respectively, then the final output is $f(x, p, b, h) + x$. This is sensible in light of the high degree of similarity between the misaligned entries and the correctly-aligned entries; in fact, we found that for the first 3 million training examples for the key network for DistilGPT2 (see section 4), the mean square difference between the input and label is only 0.0032. The implication of the residual connection is that the model is now predicting the discrepancy between the stale and new entries, rather than completely deriving the aligned entries, thus making its task easier.

Moreover, referencing StreamingLLM's implementation (mit-han-lab, 2024), we developed a generation code in which one can produce model outputs using KV-Align, StreamingLLM, and attention recomputation methods for easy comparisons. A number of evaluations on the speed and quality of generation can be found in section 5.

Figure 2. Neural network for aligning keys and values after token evictions.



4 Implementation

We start with the remark that our full implementation can be found in the footnoted repository[8].

We trained key and value networks for two LLMs: DistilGPT2 with 124M parameters (distilbert, 2024) and SmolLM2 with 135M parameters (Allal et al., 2024). Note that we opted to use small models due to limitations in available compute. Recall that the former is absolute position-based and the latter is RoPE-based. Despite the differences in positional encoding, the training data is obtained in an equivalent way for both models. Namely, we begin by collecting over 50,000 tokens from Wikitext-103-raw-v1's train split[9] (Merity et al., 2016) from Hugging Face (Wolf, 2019). Although the number of tokens seems small, recall that the key and value networks are not trying to learn any

[7] One reference for the general importance for such input-to-output residual connection is (Karpathy, 2024).

[8] https://github.com/eecs585projectorg/kv-align

[9] The choice of Wikitext dataset comes from its usage for perplexity calculation in StreamingLLM repository (mit-han-lab, 2024).

deep information about language, but rather learn how to realign the positional data in the keys and values, explaining why KV-Align is not so dependent on having many input tokens. Moreover, the number of training examples of the networks is not equal to the number of input tokens taken from Wikitext, but rather is the resulting number of (KV-dimension)-shaped vectors in the KV-cache; in fact, our approach described in the following paragraph results in over $10,000,000$ training examples just for the key network for DistilGPT2.

Afterwards, we formed groups of $N = 256$ tokens.[10] We invoke the forward method of our LLM and record the contents of the KV-cache for each head in each attention block. Following the attention sink idea in StreamingLLM (section 2.3.3), we never evict the first four keys/values for each head in each block in the KV-cache. Hence to generate the labels, we evict the fifth token to obtain a shortened input of size 255, and then repeat the procedure of saving the KV-cache for this input. Now for each of the final 251 positions, we have saved the corresponding keys/values before the token eviction (i.e., misaligned) as well as key/value following the token eviction (i.e., aligned). As explained in section 3, the input to the network now consists of the misaligned key/value as well as the position index (ranging from $0-251$ which assigns index from sixth token to the final token in that order), head index (ranging from $0-11$ for DistilGPT2) and block index (ranging from $0-5$ for DistilGPT2). The categorical indices are transformed into embedding vectors of dimensions $32, 4, 8$, respectively[11], and then they are concatenated with the misaligned key/value vector of dimension 64 (which equals the KV-dimension for both LLMs) to yield a 108-dimensional input vector. Finally, the label of the training data is given by the aligned key/value.[12]

For DistilGPT2, we used the mean squared error loss and AdamW optimizer with weight decay 0.01. Inspired by (PyTorch, 2024), we performed a hyperparameter tuning on the hidden layer sizes l_1, l_2 (where l_1 is the size of the layer that follows the input), as well as the learning rate of AdamW and the batch size. We used the ray tune package for hyperparameter tuning (Liaw et al., 2018) with ASHAScheduler (Li et al., 2020). We train the model for a maximum of 3 epochs. The results are summarized in Table 1, in which one epoch equals 5 iterations due to the chosen frequency of reporting the validation loss (in particular, 15 iterations is the maximum number of iterations). Observe that the final loss is reasonably comparable among different choices of hyperparameters, suggesting that with the input features described above, the exact values of hyperparameters are inessential, as long as they are reasonable numerical values. As a result, for both key and value networks for SmolLM2, we do not conduct hyperparameter tuning, but rather use the custom-chosen batch size of 512, learning rate of 0.0007, and hidden layer sizes of 128. We observed that the model tends to train very rapidly with validation loss stabilizing within one hour of training.

Table 1. Hyperparameter tuning results for DistilGPT2. The columns indicates trial number, $l1$, $l2$, batch size, learning rate, iterations and validation loss.

Trial	$l1$	$l2$	Batch	Rate	Iter	Loss
Key Network						
0	128	256	64	0.0531	15	0.0029
1	64	128	8192	0.0198	15	0.0022
2	32	64	8192	0.0006	15	0.0021
3	64	32	8192	0.0003	4	0.0027
4	64	256	32	0.0003	15	0.0021
5	256	32	4096	0.0142	1	0.0032
6	256	32	128	0.0005	4	0.0024
7	256	64	1024	0.0008	15	0.0019
8	32	256	512	0.0277	1	0.0029
9	32	128	32	0.0162	1	0.0031
Value Network						
0	64	128	256	0.0006	15	0.0014
1	256	32	4096	0.0001	1	0.0015
2	256	64	64	0.0680	1	0.0016
3	64	256	64	0.0027	15	0.0014
4	128	256	64	0.0007	1	0.0016
5	32	32	256	0.0008	1	0.0015
6	256	64	16384	0.0006	4	0.0015
7	64	32	8192	0.0089	4	0.0015
8	128	32	1024	0.0005	1	0.0015
9	32	256	16384	0.0002	1	0.0016

5 EVALUATION

All the training and evaluations were conducted on CloudLab c240g5 cluster equipped with two Intel Xeon Silver 4114 10-core CPUs, 192GB ECC DDR4-2666 RAM, and NVIDIA 12GB PCI P100 GPU (Dmitry Duplyakin et al., 2019; CloudLab, 2024a).

We start by presenting results on perplexity and latency of KV-Align on DistilGPT2, following the evaluation framework of the StreamingLLM authors (Xiao et al., 2024).[13]

[10] This sets a limit on the maximum cache size we can support, and a future work can experiment with generating the training data with larger N. Note that N must be at most the context length of the model.

[11] We arrived at these numbers by roughly following dimension size heuristics like (knesgood, 2024).

[12] For facilitating coding tasks like generating skeleton code for neural network training or matplotlib plots, we made use of ChatGPT (OpenAI, 2024).

[13] In particular, our Figure 4 is based on the analogous figure in the paper with the addition of the relevant data for KV-Align.

5.1 Perplexity

Perplexity is a standard measure for the next-token predictive ability of a language model evaluated on a chosen dataset (Hugging Face, 2024b), with lower perplexity indicating better ability; perplexity is computed by taking the mean of *log perplexity* over a dataset, where log perplexity is given by the negative log likelihood of the correct label based on the output multinomial probability distribution of the model from preceding tokens (section 2.1).

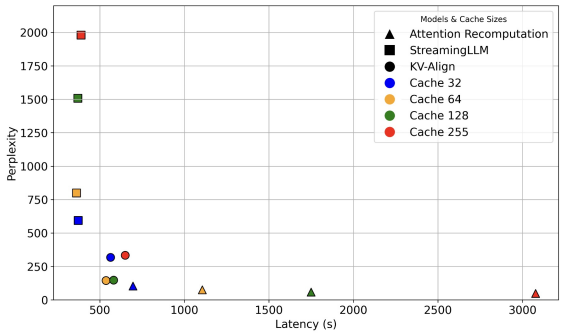


Figure 3. Perplexity plotted against its total computation time for three streaming approaches across different cache sizes for DistilGPT2. KV-Align results are the four colored circles near the bottom left corner associated with both low perplexity and compute time.

We perform perplexity calculations across cache sizes of 32, 64, 128, and 255 for attention recomputation, StreamingLLM, and KV-Align for DistilGPT2 using 20,000 tokens taken from the test split of Wikitext-103-raw-v1 (Merity et al., 2016; Xiao et al., 2024). We also keep track of the full compute time of the perplexity evaluation for each cache size and approach. Figure 3 plots perplexity versus its compute time, illustrating that KV-Align achieves balance between predictive accuracy and computational speed, as its results (circles) are located in the lower left corner of the graph across all cache sizes. This indicates a combination of lower perplexity and moderate latency compared to the other approaches.

5.2 Latency

We measure the latency of KV-Align, StreamingLLM, and attention recomputation for DistilGPT2 with cache sizes of 32, 64, 128, 255 by taking the average latency from repeating the following trial 3 times: We input a chosen prompt from MT-Bench question dataset (Zheng et al., 2023; lm-sys, 2024) and record the average time of generation per token *after* the first token for a total of 5000 tokens. We do not include the time to first token (TTFT) (Liu et al., 2024; CloudLab, 2024b) because we only care about streaming latency, not the processing time of initial prompt. The results are depicted in Figure 4.

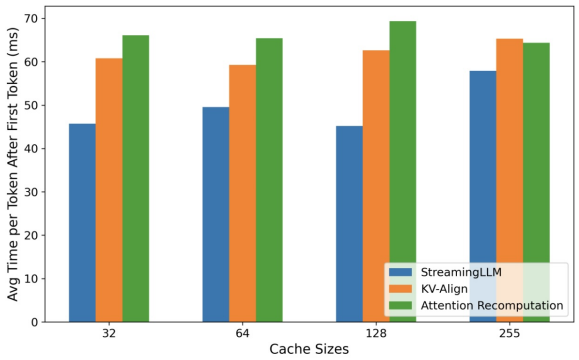


Figure 4. Average time per token after the first token of three streaming approaches across different cache sizes.

Observe that StreamingLLM demonstrates the fastest performance across all cache sizes due to its lightweight attention computation. KV-Align generally follows next. Attention recomputation is the slowest for nearly all cache sizes of 32, 64, and 128, as it recomputes all attention relationships with quadratic time complexity, leading to significantly higher computational overhead (Xiao et al., 2024). In conjunction with the perplexity evaluation, Figure 4 further emphasizes that KV-Align achieves a good balance by maintaining reasonable latency while showing lower perplexity compared to StreamingLLM.

5.3 Resistance to Distribution Shifts

We also conducted a comparison of perplexity and its compute time for Everyday Conversations datasets (HuggingFaceTB, 2024a;b) to examine the effect of using a very different dataset for evaluation than that used for the training.[14] As shown in Table 2, KV-Align again consistently achieves significantly lower perplexity than StreamingLLM while maintaining reasonable latency across all cache sizes. The results signify that it is not at all necessary to exert effort at the training stage of the key and value networks to obtain training data that is both massive in size and widely generalizable.

Table 2. Perplexity and Compute Time for Everyday Conversations Dataset with 5000 tokens for DistilGPT2 for Cache Sizes of 32, 64, 128, 255

Model	Metric	32	64	128	255
StreamingLLM	Perplexity	176.80	285.86	589.93	769.85
	Latency (s)	91.63	102.83	105.89	108.70
KV-Align	Perplexity	30.82	42.81	54.93	156.35
	Latency (s)	143.48	154.76	164.37	182.92

[14]This idea of checking for resistance to out-of-distribution dataset is inspired by a similar evaluation in (Lin et al., 2024).

5.4 Results on a RoPE-based Model

Table 3 compares the perplexity and compute time of StreamingLLM and KV-Align across different cache sizes on SmolLM2. The results indicate that KV-Align achieves comparable perplexity to StreamingLLM on RoPE models, showing that the two are orthogonal approaches. Although it is true that compute time is longer for KV-Align, we show in the next section that KV-Align sometimes has more readable outputs. Moreover, we discuss an approach to reduce the compute time for KV-Align in section 6.

Table 3. Perplexity and Latency Comparison for StreamingLLM and KV-Align on SmolLM2 for Cache Sizes of $32, 64, 128, 255$

Model	**Metric**	**32**	**64**	**128**	**255**
StreamingLLM	Perplexity	36.48	26.37	20.89	137.13
	Latency (s)	1819.96	1690.35	1753.18	1796.34
KV-Align	Perplexity	36.32	26.65	23.96	17.42
	Latency (s)	2414.28	2453.45	2537.71	2657.13

5.5 Readability

We evaluate the output readability for KV-Align by emulating an analogous approach in (tomaarsen, 2023a): we divide the LLM's output into chunks of 10 NLTK tokens[15] according to the *Words* dataset (Bird et al., 2009). Similarly to (tomaarsen, 2023a), we define a chunk as "readable" if at least half of its corresponding NLTK tokens are also found in the words dataset or are terminal punctuations. We then mark all readable chunks as green and non-readable ones as red, as illustrated in Figures 5 and 6. Notably, to improve overall readability, the generation corresponding to the former figure employs a two-gram restriction, which ensures that the generated output of the model never repeats the same immediate sequence of two tokens throughout the entire output (Wolf et al., 2024). The latter figure does not employ any 2-gram restrictions. Both figures were generated by multinomial, top-20 sampling over the token output distribution (section 2.1).

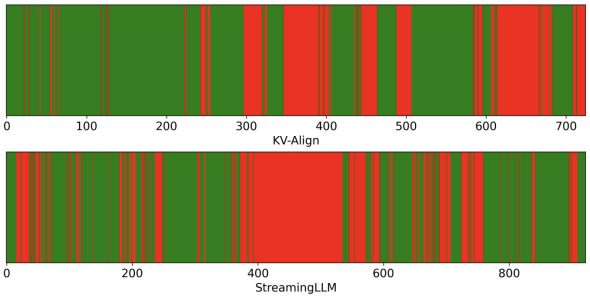


Figure 5. Readability of DistilGPT2 output, using KV-Align and StreamingLLM with two-gram restriction enabled as mentioned in section 5.5

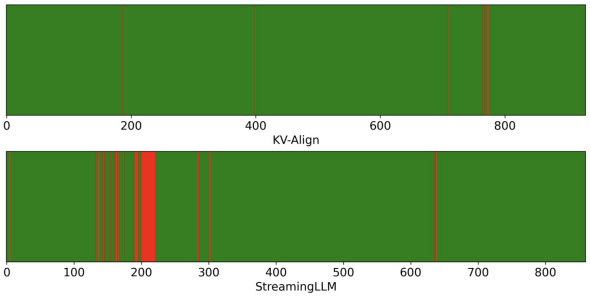


Figure 6. Readability of SmolLM2, output using KV-Align and StreamingLLM without a two-gram restriction.

6 Future Research

A limitation of our key and value networks is that they were trained specifically to predict updated cache entries for eviction of a single token. As a result, the key and value networks are applied to the entire KV-cache every time a single token is generated after the cache overflows. One improvement would be to retrain a model to allow for a batch eviction of up to N tokens and then generate N tokens in a row without any intermediate calls to the key and value networks[16]. A fix can be obtained by using a training data based on N-token evictions rather than a single token eviction. However, as this results in a more difficult realignment problem for the key/value network, it is possible that solving this adequately could require a larger model and thus result in a higher latency.

Additionally, we are limited in our current evaluation to a cache size of at most 255 due to the fact that our training data is based on 256-token chunks of Wikitext dataset. By recreating the training data to use a larger chunk size, we can better explore possibilities in substantially increasing the cache size, which could yield more interesting and practical results for future research.

Acknowledgements

Filippo Michelis, a statistics PhD student at the University of Michigan, gave some valuable personal advice for neural network training and reacted to the final version of the project. We also thank Prof. Mosharaf Chowdhury and Insu Jang for being great instructors throughout the semester and giving astute feedback on our project.

[15] Natural Language Toolkit (NLTK) (Bird et al., 2009) is a python library aimed at enhancing NLP data processing, and can tokenize strings according to given datasets.

[16] A similar sort of improvement suggestion in the context of sliding window approach was made by a reviewer of StreamingLLM in (StcB, 2024) as well as Michelis (see acknowledgements).

REFERENCES

Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebrón, F., and Sanghai, S. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

Allal, L. B., Lozhkov, A., Bakouch, E., Blázquez, G. M., Tunstall, L., Piqueres, A., Marafioti, A., Zakka, C., von Werra, L., and Wolf, T. SmolLM2 - with great data, comes great performance, 2024.

Bird, S., Klein, E., and Loper, E. *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.

Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E., Lee, P., Lee, Y. T., Li, Y., Lundberg, S., Nori, H., Palangi, H., Ribeiro, M. T., and Zhang, Y. Sparks of artificial general intelligence: Early experiments with gpt-4, 2023. URL https://arxiv.org/abs/2303.12712.

CloudLab. Hardware, 2024a. https://docs.cloudlab.us/hardware.html.

CloudLab. Metrics, 2024b. https://docs.nvidia.com/nim/benchmarking/llm/latest/metrics.html.

distilbert. DistilGPT2, 2024. https://huggingface.co/distilbert/distilgpt2.

Dmitry Duplyakin et al. The design and operation of CloudLab, 2019. https://www.flux.utah.edu/paper/duplyakin-atc19.

gante, 2023. Possible Bug with KV Caching in Llama (original) model. huggingface/transformers. [Github Issue]. https://github.com/huggingface/transformers/issues/25420#issuecomment-1775317535.

Hugging Face. Text generation strategies, 2024a. https://huggingface.co/docs/transformers/en/generation_strategies.

Hugging Face. Perplexity of fixed-length models, 2024b. https://huggingface.co/docs/transformers/en/perplexity.

HuggingFaceTB. Everyday conversations for Smol LLMs finetunings, 2024a. https://huggingface.co/datasets/HuggingFaceTB/everyday-conversations-llama3.1-2k.

HuggingFaceTB. Smoltalk, 2024b. https://huggingface.co/datasets/HuggingFaceTB/smoltalk.

karpathy. build nanoGPT, 2024. [Github repository]. https://github.com/karpathy/build-nanogpt.

Karpathy, A., 2024. Let's Reproduce GPT-2 (124M) [Video]. Youtube. https://www.youtube.com/watch?v=l8pRSuU81PU.

knesgood. Size of embedding for categorical variables, 2024. https://forums.fast.ai/t/size-of-embedding-for-categorical-variables/42608/2.

Li, L., Jamieson, K., Rostamizadeh, A., Gonina, E., Ben-Tzur, J., Hardt, M., Recht, B., and Talwalkar, A. A system for massively parallel hyperparameter tuning. *Proceedings of Machine Learning and Systems*, 2:230–246, 2020.

Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J. E., and Stoica, I. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

Lin, J., Tang, J., Tang, H., Yang, S., Chen, W.-M., Wang, W.-C., Xiao, G., Dang, X., Gan, C., and Han, S. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of Machine Learning and Systems*, 6:87–100, 2024.

Liu, J., Wu, Z., Chung, J.-W., Lai, F., Lee, M., and Chowdhury, M. Andes: Defining and enhancing quality-of-experience in llm-based text streaming services. *arXiv preprint arXiv:2404.16283*, 2024.

Liu, Z., Wang, J., Dao, T., Zhou, T., Yuan, B., Song, Z., Shrivastava, A., Zhang, C., Tian, Y., Re, C., et al. Deja vu: Contextual sparsity for efficient llms at inference time. In *International Conference on Machine Learning*, pp. 22137–22176. PMLR, 2023.

lm-sys. Fastchat, 2024. [Github repository]. https://github.com/lm-sys/FastChat/blob/main/fastchat/llm_judge/data/mt_bench/question.jsonl.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models, 2016.

mit-han-lab. Efficient streaming language models with attention sinks, 2024. [Github repository]. https://github.com/mit-han-lab/streaming-llm.

MosaicML NLP Team. Introducing mpt-7b: A new standard for open-source, commercially usable llms, 2023. www.mosaicml.com/blog/mpt-7b.

NN-SVG, 2024. Neural Network Diagram https://alexlenail.me/NN-SVG/index.html.

OpenAI, 2024. ChatGPT (4o) [Large Language Model] https://chatgpt.com/.

Press, O., Smith, N. A., and Lewis, M. Train short, test long: Attention with linear biases enables input length extrapolation. *arXiv preprint arXiv:2108.12409*, 2021.

PyTorch. Hyperparameter tuning with ray tune, 2024. https://pytorch.org/tutorials/beginner/hyperparameter_tuning_tutorial.html.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Sanh, V., Debut, L., Chaumond, J., and Wolf, T. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. In *NeurIPS EMC^{2}Workshop*, 2019.

Sinha, K., Kazemnejad, A., Reddy, S., Pineau, J., Hupkes, D., and Williams, A. The curious case of absolute position embeddings. *arXiv preprint arXiv:2210.12574*, 2022.

Stanford Online, 2023. Stanford XCS224U: NLU I Contextual Word Representations, Part 3: Positional Encoding I Spring 2023 [Video]. Youtube. https://www.youtube.com/watch?v=JERXX2Byr90.

StcB. Official review of submission2631 by reviewer stcb, 2024. [OpenReview]. https://openreview.net/forum?id=NG7sS51zVF&noteId=XyOtZijfsr.

Su, J., Ahmed, M., Lu, Y., Pan, S., Bo, W., and Liu, Y. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

tomaarsen. Attention sinks in transformers for endless fluent generation, 2023a. [Github repository]. https://github.com/tomaarsen/attention_sinks.

tomaarsen, 2023b. Bigcode architecture. attention_sinks. [Github Issue]. https://github.com/tomaarsen/attention_sinks/issues/21.

Wang, X., Salmani, M., Omidi, P., Ren, X., Rezagholizadeh, M., and Eshaghi, A. Beyond the limits: A survey of techniques to extend the context length in large language models, 2024. URL https://arxiv.org/abs/2402.02244.

Wolf, T. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Le Scao, T., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. M., 2024. URL https://github.com/huggingface/transformers/tree/main/src/transformers/generation.

Xiao, G., Tian, Y., Chen, B., Han, S., and Lewis, M. Efficient streaming language models with attention sinks. *ICLR*, 2024.

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E., et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36: 46595–46623, 2023.