Intelligently Managing KV-Cache with Reinforcement Learning

Group 36: Jingwen Gu (jg2369)

1 Introduction

1.1 Background

KV-caches are an important component of transformer models. During inference, each new token is generated by computing the attention between the current query Q_i and the keys K_j and values V_j of all previous tokens $j = 0, 1, \cdots, i-1$. To avoid redundant computation, KV-caches store the computed K_j and V_j for decoding future tokens. While accelerating the decoding process, KV-caches become a major bottleneck for long-context language modeling, since they impose a linear spatial complexity on the sequence length, and as a result most transformer LLMs have a hard-coded maximum context length beyond which the model behavior could deteriorate.

Consequently, compressing and merging the KV-cache has been a hot topic for language modeling research. The benefits from KV-cache compression is two-fold:

- If the cache size remains the same, the language model is allowed to generate far more tokens than previously allowed without significantly sacrificing performance;
- If the token length remains the same but the cahce size is reduced, the language model can operate with far smaller memory footprints while retaining the same performance.

There has also been numerous attempts to address this bottleneck by compressing the KV-cache, including FastGen[1], KVMerger[2], and KVQuant[3]. However, these methods are all fairly specific to select use cases, and our intuition is that by training a policy using Reinforcement Learning, we could better encapsulate the KV management problem and devise a more generic solution.

2 Problem

2.1 Problem Formulation

The project aims to develop an alternative solution to this bottleneck by finding a way to intelligently manage the KV-cache with reinforcement learning. The main observation is that a large number of tokens are either less informative (such as certain propositions, modal verbs, the word "the") or become less and less relevant for future tokens. If the KV matrices for these tokens are released from the KV-Cache, effectively "forgetting" these tokens, the impact on inference is less pronounced. However, one should be strategic about which tokens to "forget" and when to do so, and the decision to "forget" affects both the context and available tokens to release in future steps. Thus, this KV-cache management problem can be framed as a Markov Decision Problem $M = (\mathcal{S}, \mathcal{A}, P, r, H, \rho)$ where:

- State $\mathcal{S}$ represents the sequence of tokens $t_{<i} := t_0, t_1 \cdots, t_{i-1}$ generated so far and remaining tokens $J \subseteq \{0, 1, \cdots, i-1\}$ in the KV-cache;
- Action $\mathcal{A}$ represents the token to release from the KV-cache, which is some integer $j \in J$;
- Transition P maps the current (s, a) to the next state s'. Concretely, its complexity lies in the next generated token t_i as a function of the already-generated sequence and the remaining

cache, i.e. the entire transformer model itself. Since this is a highly complex function with large state-action space, one might need to approximate it with simpler function classes, thus rendering this MDP a Reinforcement Learning problem;

- Reward r pertains to the quality of the generated token, which can be measured by metrics such as perplexity and other task-specific criteria;
- The horizon H represent the maximum length of the generated text;
- The initial distribution ρ allows us to sample the specific input given to the language model, such as a paragraph to summarize, a question to answer, etc.

We solve this MDP using Proximal Policy Optimization (PPO), which is a widely-used heuristic for Natural Policy Gradient (NPG), a variant of Policy Gradient that is constrained by KL-divergence from an initial policy π_0.

2.2 Hypothesis

The KV-cache can be intelligently managed by using PPO to optimize a policy $\hat{\pi}$ that selects tokens to "forget" conditioned on the generated sequence $t_{<i}$ and the currently cached tokens J. Concretely, rolling out $\hat{\pi}$ should yield superior performance to baseline methods such as FastGen and KVMerger on select use cases.

2.3 Related class concepts

MDP, Reinforcement Learning, Policy Gradient, Function Approximation

3 Approach

In this section, we discuss the implementation details of the environment for KV-Cache management and the application of the PPO algorithm to different variants of the environmnet. In particular, solving the KV-cache MDP M using PPO requires a nuanced treatment of the state space $\mathcal{S}$ and the reward r. As mentioned in previous sections, the KV-cache can be excessively large, which can make optimization intractable if both the KV-cache and generated tokens are parameterized as state variables. On the other hand, the reward function r can vary from task to task, but training a different policy model for each specific task may be computationally expensive and undesirable for generic language models. Hence in this section, we develop different variants of the state representation and different methods to compute reward to ensure the feasibility of PPO in solving the KV-cache management problem.

3.1 State Representation

To reduce the size of the state space $\mathcal{S}$, we explore two variants of the original MDP where the state space is restricted to only the remaining KV-cache or only to the generated token IDs, effectively resulting in two new MDPs M_{KV} and M_{ID}. Conceptually, both representations should convey abundant information useful for choosing the token to release from the KV-cache:

- M_{KV}: During inference, the transformer model takes in the K, V matrices as the context representation. Therefore the actual values of the K, V matrices are analogous to word embeddings are the most direct way to inform the model on what was generated so far. It can be imagined that similar words would have similar K, V representations, and by analyzing these matrices the policy model might thus learn to discern less informative words such as prepositions, or that are less relevant to the current context.
- M_{ID}: Even if we do not pass in the actual K, V matrices, there is a bijective mapping between tokens and their IDs. In particular, certain words like "the", "is", and "a" that are especially uninformative in the long run may be easily identified by the model using IDs, so the model may learn to put stronger weights that lean towards choosing these words. However, since IDs are randomized, what the model learned about one particular ID cannot generalize towards other IDs, which may make it harder for the model to learn more nuanced tricks to KV management.

We compare the dimension of each state tensor from the two MDPs:

MDP	State tensor dimension	Dimension for GPT2	Dimension for LLaMA 3.1-8B-Instruct
M_{KV}	$[\texttt{n\_layers}, 2, B, H, \texttt{n\_heads}, D]$	[12,2,B,1024,12,64]	[32,2,B,2048,32,128]
M_{ID}	$[B, H]$	[B,1024]	[B,2048]

Here B is the batch size, H is the maximum length, D is the dimension of the KV matrices, `n_layers` is the number of layers of the transformer model, and `n_heads` denotes the number of attention heads. Note that even though the H dimension would be replaced by the current length the the generated text during inference, we have to pad it to H such that the state space dimension remains the same for all steps. As shown in the table, M_{KV} requires a much larger state space than M_{ID}, and for the LLaMA 3.1-8B-Instruct model the M_{KV} state space is already rather intractable.

Hence there is a trade-off between expressiveness and tractability when choosing between the two MDPs. For smaller models such as GPT2, we will experiment with both MDPs; whereas for larger models such as LLaMA 3.1-8B-Instruct, we will primarily focus on M_{ID}.

3.2 The PPO Algorithm

In this paper, we do not modify the implementation of the PPO Algorithm, so in essence PPO is used as a black-box oracle for policy optimization. Consequently, one can easily replace PPO with their favorite policy optimization algorithm. For our experiments, we use the PPO class from Stable-Baselines3, and hyperparameters will be given in the Results section. With the optimization algorithm ascertained, our main work lies in implementing the environments for each variant of KVM, which we discuss in the following subsections.

3.3 PPO Training with Sparse Reward

In specific language modelling tasks such as summarization, sentiment analysis, and chatbot response generation, an overall loss function for the entire generated text is available to gauge the performance of the language model. It is then natural to use a sparse reward function that assigns the overall loss to the last step of each trajectory, and 0 everywhere else.

We thus define the environment KVM-Sparse, which is an abstract interface that could be implemented with different sparse reward functions for different specific tasks. It has the following behaviors:

1. When resetting the environment, it randomly samples a sequence of tokens t_0 from a tokenized dataloader, and uses the language model to autoregressively generate until it reaches the sequence length C, where C is the cache size. This is because we are only interested in the behavior of the language model when the number of tokens exceeds the KV-cache size.
2. The environment maintains the "actual" token IDs $t_{\leq h}$ which can be arbitrarily long, and the "managed" token IDs $\hat{t}_{\leq h}$ and KV which have the length of C.
3. As mentioned in section 2.1, the action a corresponds to the index of the KV-cache to release. When stepping the environment, the corresponding a-th element in both $\hat{t}_{\leq h}$ and KV are removed. We guarantee that a is not chosen to be the most recently generated token to avoid the language model falling into a loop.
4. The language model then generates using the most recently generated token $t[-1]$ and KV. Note that the last token is provided because by construction of the transformer architecture, the current KV includes all generated context up to the most recently generated token.
5. The next token t_h is greedily chosen by the log probability output from the language model, and is concatenated to both the actual tokens t and the managed tokens $\hat{t}$.

Below we present the pseudo-code of the `reset` and `step` functions of KVM-Sparse:

Algorithm 1: KVM-Sparse Environment

```
Class variables:
$f(t, KV)$: The language model as a function of the input IDs and KV-cache
$t_{\leq h}$: The sequence of token IDs that the language model has generated
$\hat{t}_{\leq h}$: The sequence of generated token IDs that remain in the KV-cache
$\hat{KV}$: The remaining KV-cache
$h$: The current step
$C$: The KV-cache size

function reset():
    $t_0 \sim \rho$
    $t_{\leq C}, KV \leftarrow f(t_0, \emptyset)$
    $\hat{t}_{\leq C} \leftarrow t_{\leq C}$
    $h \leftarrow C$

function step(s,a):
    $KV \leftarrow KV[: a] \cup KV[a+1 :]$
    $\hat{t}_{\leq h-1} \leftarrow \hat{t}_{\leq h-1}[: a] \cup \hat{t}_{\leq h-1}[a+1 :]$
    $\Delta, KV \leftarrow f(t[-1], KV)$ # $\Delta$ is the output logits from the language model
    $t_h \leftarrow \arg\max_{v \in V} \Delta(v)$ # Greedily decode the next token
    $t_{\leq h} \leftarrow t_{\leq h-1} \cup t_h$
    $\hat{t}_{\leq h} \leftarrow \hat{t}_{\leq h-1} \cup t_h$
    $h \leftarrow h + 1$
    $E \leftarrow \mathbf{1}(h = H \vee t = \texttt{eos\_token})$ # decide whether terminates
    return $(s(\hat{t}_{\leq h}, KV), r_h(s, a), E)$
```

Note that in the last line, $s(\hat{t}_{\leq h}, kv)$ is a function that converts the remaining token IDs and KV-cache into the state vector, which is different between M_{KV} and M_{ID}; $r_h(s, a)$ is the sparse reward function specific to the task being trained for.

In this paper, we mainly focus on summarization as an example of using sparse rewards for KVM. For text summarization tasks, a widely-used metric is ROUGE, which compares the similarity between two summaries written for the same original text, one being the model generation and another being the ground truth. There are 4 main variants of ROUGE: ROUGE-1, which checks for unigram overlap; ROUGE-2, which checks for bigram overlap; ROUGE-L, which checks for longest common subsequences, and ROUGE-LSum, which is a variant of ROUGE-L for longer summaries. Here we select ROUGE-L as the primary metric for summarization because it is the most appropriate for our expected length and quality of the generated summaries.

The sparse reward function can then be written as follows:

$$r_h^{\tau}(s, a) = \begin{cases} 0 & h < |\tau| - 1 \\ \text{ROUGE}(s) & h = |\tau| - 1 \end{cases} \tag{1}$$

Note that here we are comparing the current timestep h with $|\tau|$ and not H, where $|\tau| \leq H$ is the length of the current trajectory. This is because even though we could pad all trajectories to the same length H, running the language model to generate the extra padded tokens at the end is an unnecessary computational overhead, so we could simply terminate the trajectory when the `eos_token` is generated.

3.4 PPO Training with Teacher Forcing

Despite its simplicity, KVM-Sparse suffers from the drawback of having a different reward for each particular task. It would then require training a different policy model for each task and, if the actual application consists of a combination of tasks such as translation-summary, switching between policy models during inference. This defeats the purpose of generic language models.

Hence we turn to the language modeling literature, where it has been proven that supervised finetuning provides large language models with reasonable generalization abilities, and numerous papers have studied zero-shot or few-shot usages of language models. This is because by minimizing the cross entropy loss between the generated and ground truth tokens, the language model learns to imitate the human speakers so well that they develop sufficient overall language skills to excel in more specific tasks.

Similarly, we explore the idea of using PPO to train a policy model that manages the KV-cache for all tasks in general. However, using the token-level cross entropy loss is problematic for autoregressive training, since once the language model generates a different token than the ground truth, all future tokens will be affected, and hence it becomes hard to quantify the difference between the generated text and the ground truth trajectory. On the other hand, the trajectory space of language modelling is so large that one cannot hope to have a large enough dataset to cover all possible previously-generated tokens $t_{\leq h}$ as states, rendering imitation learning algorithms like Behavior Cloning unfeasible.

We thus borrow the technique of Teacher Forcing from the language modeling community. When training with teacher forcing, the language model f is allowed to generate one next token t_h, and a token-level cross-entropy loss is computed between the generated token and the corresponding ground truth token t_h^*. To avoid the trajectory deviating from the ground truth, the model is then fed the ground truth token at the next step instead of its own generated token. This ensures that the model and the ground truth have the same input tokens at each step, and the overall cross entropy loss is defined as

$$L(f, t_{\leq H}^*) = \exp\left(-\frac{1}{H}\sum_{h=0}^{H-1} \log \mathbb{P}_f(t_h^* | t_{<h}^*)\right) \tag{2}$$

In the same vein, we define a second environment KVM-TF with teacher forcing for PPO training:

1. The reset function is the same as in KVM-Sparse and the environment still maintains the "actual" token IDs $t_{\leq h}$, and the "managed" token IDs $\hat{t}_{\leq h}$ and KV. In addition, we also store the ground truth token IDs $t_{\leq H}^*$.
2. When stepping the environment, the corresponding a-th element in both $\hat{t}_{\leq h}$ and KV are removed and the language model then generates using the most recently generated token $t[-1]$ and KV.
3. Different from KVM-Sparse, the next token t_h is never actually decoded; instead, we compute the token-level cross-entropy reward $r_h(\Delta, t_h^*) = -\log \Delta(t_h^*)$. This reward is strictly negative and larger value indicates better performance. We then concatenate the ground truth token t_h^* to both the actual tokens t and the managed tokens $\hat{t}$.

Because the environment design is similar to KVM-Sparse, we leave the pseudo-code for KVM-TF to Appendix. It should be noted, however, that using teacher forcing violates assumptions for Reinforcement Learning; namely, during inference the policy model will not have access to the ground truth tokens, yet in the training environment they do. This proved not to be an issue for language modeling in general, but it is unclear whether this is also the case for KVM.

3.5 PPO Training with Distillation

To simultaneously avoid violating RL assumptions and ensure generalizability, we develop a third approach named KVM-Distill which generates the ground truth tokens alongside the predicted tokens when rolling out the policy, effectively distilling from the inferring with the full KV-cache to inferring with the managed KV-cache. This is structurally similar to the first environment KVM-Sparse, but for each step we do two model inferences instead of one: once on the managed token ids $\hat{t}_{\leq h}$, once on the actual token ids $t_{\leq h}$, both with the same language model, and obtain logits $\hat{Delta}$ and Δ respectively. We then make the reward function a similarity measure between the two logits $r(s, a) = \texttt{sim}(\hat{\Delta}, \Delta)$, which can be implemented by a token-level cross entropy loss or KL-divergence, among other plausible metrics:

$$r(s,a) = \begin{cases} \log \hat{\Delta}(\arg\max_{v \in V} \Delta(v)) & \text{(CE)} \\ D_{KL}(\hat{D} || D) = \sum_{v \in V} \frac{e^{\hat{\Delta}(v)}}{\sum_{v'} e^{\hat{\Delta}(v')}} \left(\hat{\Delta}(v) - \Delta(v) - \log\left(\frac{\sum_{v'} e^{\hat{\Delta}(v')}}{\sum_{v'} e^{\Delta(v')}}\right)\right) & \text{(KL)} \end{cases} \tag{3}$$

Similar to the previous subsection, we leave the pseudo-code in the appendix. It should also be noted that even though KVM-Distill avoids the pitfalls of the other two approaches, it doubles the amount of model inference, and its reliance on using the same language model for inference on the actual unabridged tokens means that it is unable to learn to generate more tokens than the language model's built-in horizon allows. However, it still provides the benefit of generating with smaller memory footprint (section 1.1, point 2), and we hypothesize that we could use a larger model for generating ground truth logits. This can be easily implemented for the Cross-Entropy reward, and also for the KL-reward provided that the two models share the same tokenizer. Nevertheless, this would further increase the computational overhead by involving a larger model in the inference loop, so we shall leave this modification for future work.

4 Results

4.1 Experiment Setup

Since the proposed method is intended to be generic for all language modeling tasks (with optional task-specific finetuning), the langauge model, dataset and platforms will be selected to reflect common language modeling use cases. The setup are as follows:

Language model: GPT-2-medium, Llama-3.1-8B-Instruct to explore the effect on models of different parameter sizes. Larger models will be hard to load into a personal GPU and is hence out of the scope of this project;

Policy Model: because the policy model needs to be run whenever a token is decode, it must be lightweight and fast in order to minimize the computation overhead and latency. Because the state-action space is huge, it is natural to functionally approximate this policy model by a small MLP. Here we use a 3-layer MLP for both the actor and the critic; the actor MLP has hidden size 128 for all three layers, and the critic MLP has hidden size 256 for all three layers.

Datasets:

- Wikitext-v2[4]: a generic corpus extracted from Wikipedia. Collecting rollouts of the policy on this dataset provides a good measure of the general text-generation ability of the language model under the KV-cache management policy.
- SAMSum[5]: a human-annotated dataset for dialogue summarization tasks. Each entry consists of a dialogue text consisting of 3-5 turns between 2-3 characters, and a ground-truth summary of the dialogue. We chose this dataset since its dialogue length is similar to the cache size we set for KVM, and requires little expert knowledge about sophisticated backgrounds to summarize, which is ideal considering that we are prioritizing smaller language models.

We conduct two main classes of experiments:

- General language modeling ability tests. This is done in a similar fashion as the Teacher-Forcing environment: the policy model manages the KV-cache, and the language model uses the abridged KV-cache and input tokens to perform inference on the wikitext dataset. A per-token cross-entropy loss is computed between the output logits and ground-truth token, and the ground truth token is then appended to the KV-cache and input tokens for the next generation step. The average token-level cross entropy and perplexity values are recorded. If the model achieves good scores in this experiment, it means that the model will be able to perform well in general language modeling tasks.
- Summarization tests. This is done on the SAMSum corpus, and is rolled out in the KVM-Sparse environment, i.e. the next token decoded from the output logits on the abridged KV-cache and tokens are appended to the generation results, and the model continues to generate autoregressively conditioned on the abridged input. The generated summary is compared with the ground truth summary from the SAMSum corpus using the ROUGE metric, and we report all four ROUGE metrics for each algorithm.

For each experiment, we roll out the following policies:

- KVM-Sparse: This is the policy trained using the KVM-Sparse environment for summarization specifically.
- KVM-HF: This is the policy trained using the KVM-HF environment with teacher forcing for general language modeling capabilities.
- KVM-Distill-KL: This is the policy trained using the KVM-Distill environment where the similarity metric for distillation is KL-Divergence.
- KVM-Distill-PPL: This is the policy trained using the KVM-Distill environment where the similarity metric for distillation is Cross Entropy Loss, which is essentially related to Perplexity.

We also include the following baselines for comparison:

- Random: This policy simply randomly chooses an index and releases the KV-Cache there.
- Sliding Window: This policy always releases the oldest element in the KV-Cache.

Hence we have 2 experiment classes, 2 language model choices, and 6 KV-cache management policies, resulting in 24 different experiment runs.

4.2 Training Process

For the PPO training process, we set the total steps to be 1500000, where each step corresponds to a generated token; we also do validation and policy optimization every 5000 steps. We set the `ent_coef` parameter to 0.01 to allow for better exploration-exploitation trade-off, and we set the learning rate to $3e-4$. For the KVM-TF policy model, however, due to the huge variance in rewards, we used a custom learning rate setter which reduces the learning rate with higher rewards shown below, thus helping the model converge.

$$lr = 10^{\min\{-5,-10-R/100\}} \tag{4}$$

We show the reward curves for some of the policy models as they were trained:

4.3 Experiment Results

We report the experiment results for both experiments below:

Table 1: PPL scores on general language modelling

Algorithm	Environment Reward	Per-token CE Loss	Perplexity
GPT-2			
Random	-853.1944	-6.4669	1666.0275
Sliding Window	-957.7592	-8.6928	16142.2238
KVM-HF	-416.3925	-4.0582	68.8803
KVM-Distill-KL	-597.7086	-6.5830	2750.1760
KVM-Distill-PPL	-407.4126	-4.8834	405.6477

It can be seen that all four of our policies upper-bound the two baseline methods by a significant margin in all experiment settings. In particular, KVM-HF and KVM-Distill outperforms KVM-Sparse in the summarization test, which shows that these methods actually do succeed in achieving high generalizability; on the other hand, by sparsifying the reward, it may have become too difficult for the agent to improve its policy.

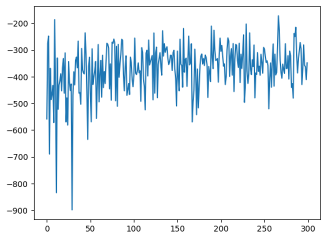
Figure 1: Training reward of KVM-TF with GPT2

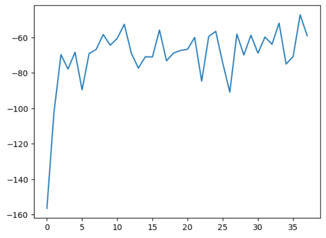
Figure 2: Training reward of KVM-Distill-PPL with LLaMA-3.1-8B-Instruct

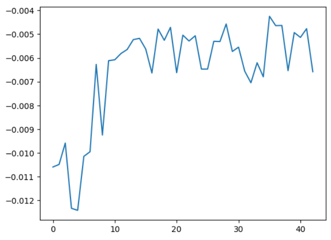
Figure 3: Training reward of KVM-Distill-KL with GPT2

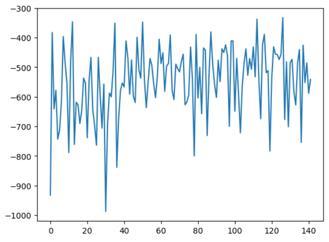
Figure 4: Training reward of KVM-TF with LLaMA-3.1-8B-Instruct

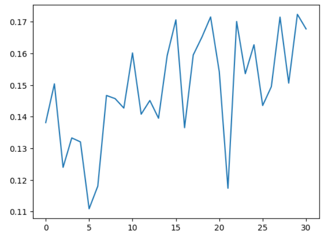
Figure 5: Training reward of KVM-Sparse with LLaMA-3.1-8B-Instruct

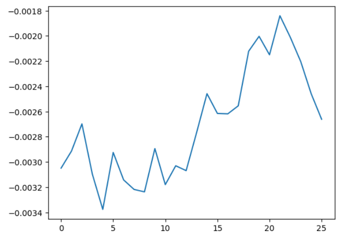
Figure 6: Training reward of KVM-Distill-KL with LLaMA-3.1-8B-Instruct

Table 2: ROUGE scores on summarization tests

Algorithm	Rouge-1	Rouge-2	Rouge-L	Rouge-LSum
GPT-2				
Random	0.2262	0.0631	0.1681	0.2093
Sliding Window	0.2203	0.0693	0.1744	0.2080
KVM-Sparse	0.2437	0.0840	0.1999	0.2297
KVM-HF	0.2518	0.0927	**0.2147**	0.2367
KVM-Distill-KL	**0.2721**	0.0930	0.2077	**0.2646**
KVM-Distill-PPL	0.2627	**0.0988**	0.1962	0.2401
LLaMA-3.1-8B-Instruct				
random	0.1509	0.0576	0.1252	0.1461
Sliding Window	0.1936	0.0704	0.1545	0.1825
KVM-HF	**0.2968**	**0.1305**	**0.2240**	**0.2692**
KVM-Distill-KL	0.2308	0.0575	0.1522	0.2093
KVM-Distill-PPL	0.2446	0.0924	0.1791	0.2264
KVM-Sparse	0.2548	0.0910	0.1882	0.2428

References

[1] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, and Yuxiong He. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-

inference, 2024.

[2] Zheng Wang, Boxiao Jin, Zhongzhi Yu, and Minjia Zhang. Model tells you where to merge: Adaptive kv cache merging for llms on long-context tasks, 2024.

[3] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W. Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization, 2024.

[4] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016.

[5] Bogdan Gliwa, Iwona Mochol, Maciej Biesek, and Aleksander Wawer. Samsum corpus: A human-annotated dialogue dataset for abstractive summarization. In *Proceedings of the 2nd Workshop on New Frontiers in Summarization*. Association for Computational Linguistics, 2019.

Appendix A

KVM-TF Pseudo-code

Below we present the pseudo-code of the `reset` and `step` functions of KVM-TF. Note that the red highlighted parts indicate important changes from KVM-Sparse.

```
Algorithm 2: KVM-TF Environment
Class variables:
$f(t, KV)$: The language model as a function of the input IDs and KV-cache
$t^*_{\leq H}$: The ground truth token IDs
$t_{\leq h}$: The sequence of token IDs that the language model has generated
$\hat{t}_{\leq h}$: The sequence of generated token IDs that remain in the KV-cache
$KV$: The remaining KV-cache
$h$: The current step
$C$: The KV-cache size
function reset():
    $t^*_{\leq H} \sim \rho, t_{\leq C} \leftarrow t_{\leq H}[: C]$
    $KV \leftarrow f(t_0, \emptyset)$
    $\hat{t}_{\leq C} \leftarrow t_{\leq C}$
    $h \leftarrow C$
function step(s,a):
    $KV \leftarrow KV[: a] \cup KV[a + 1 :]$
    $\hat{t}_{\leq h-1} \leftarrow \hat{t}_{\leq h-1}[: a] \cup \hat{t}_{\leq h-1}[a + 1 :]$
    $\Delta, KV \leftarrow f(t[-1], KV)$ # $\Delta$ is the output logits from the language model
    $r_h \leftarrow -\log \Delta(t^*_h)$
    $t_{\leq h} \leftarrow t_{\leq h-1} \cup t^*_h$
    $\hat{t}_{\leq h} \leftarrow \hat{t}_{\leq h-1} \cup t^*_h$
    $h \leftarrow h + 1$
    $E \leftarrow \mathbf{1}(h = H \vee t =$ eos_token$)$ # decide whether terminates
    return $(s(\hat{t}_{\leq h}, KV), r_h, E)$
```

KVM-Distill Pseudo-code

Below we present the pseudo-code of the `reset` and `step` functions of KVM-Sparse. Again, note that the red highlighted parts indicate important changes from KVM-Sparse.

Algorithm 3: KVM-Distill Environment

```
Class variables:
$f(t, KV)$: The language model as a function of the input IDs and KV-cache
$t_{\leq h}$: The sequence of token IDs that the language model has generated
$\hat{t}_{\leq h}$: The sequence of generated token IDs that remain in the KV-cache
$KV$: The remaining KV-cache
$h$: The current step
$C$: The KV-cache size

function reset():
    $t_0 \sim \rho$
    $t_{\leq C}, KV \leftarrow f(t_0, \emptyset)$
    $\hat{t}_{\leq C} \leftarrow t_{\leq C}$
    $h \leftarrow C$

function step(s,a):
    $KV \leftarrow KV[: a] \cup KV[a+1 :]$
    $\hat{t}_{\leq h-1} \leftarrow \hat{t}_{\leq h-1}[: a] \cup \hat{t}_{\leq h-1}[a+1 :]$
    $\hat{\Delta}, KV \leftarrow f(t[-1], KV)$ # $\hat{\Delta}$ is the output logits from the language model
    $\Delta \leftarrow f(t_{\leq h})$ # $\Delta$ is the output logits conditioned on the unabridged tokens
    $r_h(s, a) \leftarrow \texttt{sim}(\hat{\Delta}, \Delta)$ # Implementation of sim can be CE, KL, etc.
    $t_h \leftarrow \arg\max_{v \in V} \hat{\Delta}(v)$ # Greedily decode the next token from the abridged logits
    $t_{\leq h} \leftarrow t_{\leq h-1} \cup t_h$
    $\hat{t}_{\leq h} \leftarrow \hat{t}_{\leq h-1} \cup t_h$
    $h \leftarrow h + 1$
    $E \leftarrow \mathbf{1}(h = H \vee t = \texttt{eos\_token})$ # decide whether terminates
    return $(s(\hat{t}_{\leq h}, KV), r_h(s, a), E)$
```