MEMORIZING TRANSFORMERS

Anonymous authors
Paper under double-blind review

ABSTRACT

Language models typically need to be trained or finetuned in order to acquire new knowledge, which involves updating their weights. We instead envision language models that can simply read and memorize new data at inference time, thus acquiring new knowledge immediately. In this work, we extend language models with the ability to memorize the internal representations of past inputs. Despite the fact that our implementation of memory is not differentiable, we demonstrate that an approximate kNN lookup into the memory improves language modeling across various benchmarks and tasks, including generic webtext (C4), math papers (arXiv), books (PG-19), code (Github), as well as formal theorems (Isabelle). We show that the performance steadily improves when we increase the size of memory up to 131k tokens. We also find that the model is capable of making use of newly defined functions and theorems during test time.

1 INTRODUCTION

Transformers (Vaswani et al., 2017) have become one of the most widely-used neural network architectures for a variety of tasks, especially natural language processing (NLP). They have proved to be immensely scalable, achieving state of the art performance for models of all sizes, up to and including giant models with billions of parameters (Brown et al., 2020). However, although transformers scale very well with the number of layers and parameters, they scale very poorly with sequence length, and their computational and memory complexity is infamously $O(N^2)$.

Due to this limitation, training of transformers typically proceeds by chopping long documents into much shorter subsequences, which deprives the model of information about the larger context in which a passage occurs. On many tasks, such as processing of books, source code, technical papers, and theorems, the ability to attend to far-away tokens is important. In source code, for example, references to classes and functions may occur quite far from the places in which they are defined.

Attention over long sequences is also useful as a form of rapid learning. Facts and information which are stored in the form of weight matrices must be slowly trained over hundreds of thousands of training steps. By using attention, however, a model can simply *memorize* facts (e.g. function definitions) by storing them as (key, value) pairs in long-term memory, and then retrieve those facts later by creating a query that attends to them.

Memorizing transformers are motivated by our desire to use transformers for program synthesis and formal reasoning. For these tasks, the model must be able to work with large and continuously changing code repositories and knowledge bases, and it should be possible to utilize newly added code or facts immediately without the need for retraining or finetuning. Our goal is to develop forms of attention that can scale to the size of an entire code repository, or a database of theorems.

We demonstrate that a simple, effective, and scalable way to increase the size of the attention context is to use approximate k-nearest-neighbor (kNN) search into a large external memory of (key, value) pairs. There are efficient implementations of approximate kNN lookup on TPU, GPU, and CPU, including distributed implementations (Guo et al., 2020), which opens the door to extremely large external memories.

In contrast to most other work on sparse, or long-range attention (c.f. Section 2), we treat the external memory as a large "cache", and gradients are not back-propagated into the cache. This is a potential limitation, because it means that the network can only learn to *query* the external memory, and cannot

directly learn what keys and values to put into it. However, we demonstrate empirically that (key, value) pairs which are useful for local (and thus fully differentiable) self-attention are also useful for long-range attention.

Using a non-differentiable cache is critical to scalability. The keys and values are a function of model parameters, so attempting to backpropagate gradients into the external memory would necessarily involve computing all of the keys and values with the current model parameters on every training step. If the external memory is not differentiable, we can instead reuse keys and values from prior training steps. With our technique, we are easily able to scale external memory up to a sequence lengths of 131k or 262k tokens on a single device, while maintaining a reasonable step time.

We further optimize for speed and GPU/TPU memory by using the external memory only in a single layer, near the top of the transformer stack, rather than integrating it into every transformer layer, as is done with most other forms of attention. The lower levels of the transformer stack use classical dense attention, and are responsible for parsing, summarizing, and otherwise processing information in the input sequence, using only the local context in which tokens occur. Our *kNN-augmented attention layer* then stores the processed (key, value) pairs into external memory, and also issues queries to retrieve long-term memories from it. One additional dense attention layer at the top of the stack then integrates the local and long-context information.

We show that model perplexity steadily improves with the size of external memory on a variety of language modelling tasks, including C4 (long documents only), Github code repositories, PG-19 books, formal proofs in Isabelle, and arXiv math papers. We further show that models can generalize to larger memory sizes than they were trained on: models trained with a small kNN memory show gains from using a much larger memory at inference time. Finally, we show that our models are actually using external memory in the way that we had hoped, e.g. by looking up the definitions of lemmas in a theorem proving corpus.

2 Related Work

A great deal of work has been done on efficient long-range attention mechanisms; see Tay et al. (2020b) for a recent survey. Sliding windows (Beltagy et al., 2020) use a long sequence, but attend within a smaller window, thus reducing complexity to the window size, rather than total sequence length. Approximate mechanisms such as Linformer (Wang et al., 2020a), and Performer (Choromanski et al., 2021) refactor the attention matrix by using a different kernel than softmax to obtain $O(N)$ complexity. Pooling strategies such as Hierarchical 1D attention (Zhu & Soricut, 2021), and Combiner (Ren et al., 2021) apply pooling or averaging over tokens at longer distances. Sparse strategies such as Big Bird (Zaheer et al., 2020) select only a subset of tokens to attend to; Routing Transformers (Roy et al., 2021) use clustering to select the subset, while Reformer (Kitaev et al., 2020) relies on hashing. Hierarchical mechanisms (Ainslie et al., 2020) combine multiple tokens into phrases or sentences to reduce sequence length. Expire-span (Sukhbaatar et al., 2021) prunes far-away tokens that it learns are "unimportant". See Tay et al. (2020a) for a comparison of many of these approaches.

Feedback transformers (Fan et al., 2020) use a recurrent architecture, in which each token attends to the output of the final layer, rather than to the previous layer. Recurrence does not increase the size of the attention context, but it expands the *receptive field* at the cost of parallelism and training speed.

Truncated backpropagation through time (Williams & Peng, 1990) was originally introduced as a way of training recurrent neural networks (RNN) over very long sequences, when the entire sequence does not fit in memory. The sequence is chopped into segments, and after each training step, the final RNN state for the segment is saved in a non-differentiable cache, and used as the initial state on the next training step. Neural caches (Grave et al., 2017) extend the cache to contain a record of many prior hidden states, and attend over them. Transformer-XL (Dai et al., 2019) applies this technique to transformers; it caches the (key,value) pairs computed from the previous training step, and uses them as a prefix for the tokens on the next training step, which yields significant gains on long documents. Rae et al. (2020) improve over Transformer-XL by compressing the tokens before adding them to the cache. In contrast, we use a very large cache without compression, combined with an approximate kNN attention mechanism over it.

Sukhbaatar et al. (2019) make the observation that the feed-foward portion of a transformer layer functions very much like attention, if one simply replaces the relu activation with softmax. They

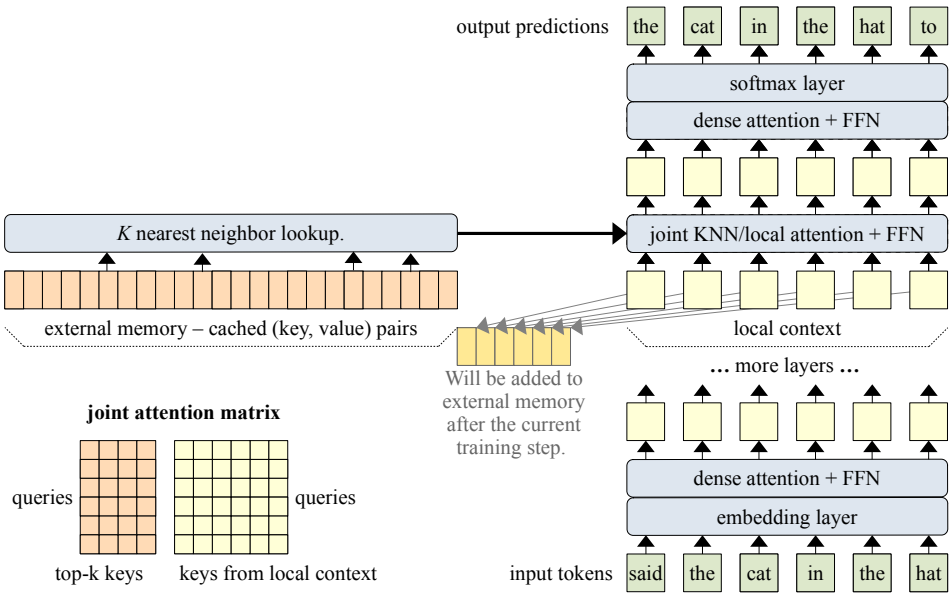

Figure 1: We extend Transformers with access to (key, value) pairs of previously seen subsequences.

implement a combined attention over both tokens from the input sequence and a learned (and differentiable) "memory". Lample et al. (2019) exploit this observation to replace the FFN with a fast kNN lookup over a much larger "memory", and achieve large gains in model accuracy without much computation overhead. (We use kNN lookup for attention, not as a replacement for the FFN.)

Non-differentiable external memory has been used in different ways by other architectures. (Khandelwal et al., 2020) run a pre-trained model over an entire corpus, and construct a large table of (key,token) pairs. They then use that table to replace the final softmax layer for token selection in the model, which results in significant improvements in language modeling.

Retrieval-augmented transformers, such as REALM (Guu et al., 2020) and MARGE (Lewis et al., 2020), and composite memory for dialog (Fan et al., 2021), retrieve documents from a knowledge base. The knowledge base is usually static and separate from the inputs and outputs of the models. Instead, we focus on language modeling using a decoder-only model, which unifies inputs, outputs, and retrievable memory.

k-nearest-neighbor lookup is a general-purpose technique that is used for a wide variety of machine learning and retrieval tasks, and many high-performance implementations are available for various architectures (Johnson et al., 2021; Guo et al., 2020). Memory-efficient Transformers (Gupta et al., 2021) replace dense attention with a kNN lookup to increase speed and reduce memory usage.

3 METHOD

The architecture of our kNN-augmented transformer is shown in Figure 1. The bulk of the model is a vanilla, decoder-only transformer (Vaswani et al., 2017). The input text is tokenized, and the tokens are embedded into vector space. The embedding vectors are passed through a series of transformer layers, each of which does dense self-attention, followed by a feed-forward network (FFN). Since this is a decoder-only language model, so we use a causal attention mask and the token embeddings of the last layer are used to predict the next token.

We split a long document up into subsequences of length 512 tokens. Each subsequence is used as the input for one training step. In contrast to standard practice, we do not shuffle the subsequences; instead, each long document is fed into the transformer sequentially, from beginning to end, as is done with Transformer-XL (Dai et al., 2019).

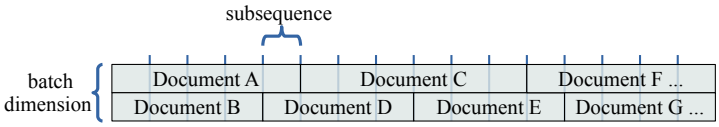

Figure 2: Our data pipeline splits documents into subsequences and packs subsequences into batches.

3.1 kNN-AUGMENTED ATTENTION LAYER

The second-to-last transformer layer is a *kNN-augmented attention layer.* This layer combines two forms of attention. The first is standard dense self-attention into the *local context*, which is the input subsequence for the current training step. The second is an approximate k-nearest-neighbor search into the *external memory*. The same queries are used for both the local context, and for the external memory. The keys and values also belong to the same distribution; after each training step, the (key, value) pairs in the local context are appended to the end of the external memory. If the document is very long, old (key, value) pairs will be dropped from the memory to make room for new ones. Thus, for each head, the external memory keeps a cache of the prior M (key, value) pairs, where M is the memory size.

The kNN lookup will return a set of *retrieved memories*, which consist of the top-k (key, value) pairs that kNN search returns for each query (i.e. each token) in the input subsequence. As with standard dense attention, we construct an attention matrix over external memory by computing the dot product of each query against the retrieved keys.

The dot-product scores for the k retrieved memories are concatenated with the scores from dense self-attention over the local context prior to applying softmax. In other words, the external memory is treated as an extension of the local context; each query attends to both the local context and external memory jointly. The output of the kNN-augmented attention layer is a weighted sum of the values retrieved from external memory, and values from the local context.

More formally, let $\mathbf{h}_i$ be the output vector for the i^{th} token. It is defined as follows, where $\mathbf{q}$, $\mathbf{k}$, and $\mathbf{v}$ are the query, key, and value vectors for the local context, $\tilde{\mathbf{k}}$, $\tilde{\mathbf{v}}$ are the key and value vectors returned from the kNN search, e_{ij} and $\tilde{e}_{ij}$ are the dot-product similarity scores, and a_{ij}, $\tilde{a}_{ik}$ are the attention weights. Notice that unlike self-attention, kNN retrieves a different set of ($\tilde{\mathbf{k}}$, $\tilde{\mathbf{v}}$) for each token; $\tilde{\mathbf{v}}_{ik}$ thus denotes the k^{th} value in the set of retrieved memories for the i^{th} token.

$$
\begin{aligned}
e_{ij} &= \frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d}} & \tilde{e}_{ik} &= \frac{\mathbf{q}_i^\top \tilde{\mathbf{k}}_{ik}}{\sqrt{d}} & s_i &= \sum_j \exp(e_{ij}) + \sum_k \exp(\tilde{e}_{ik}) \\
a_{ij} &= \frac{\exp(e_{ij})}{s_i} & \tilde{a}_{ik} &= \frac{\exp(\tilde{e}_{ik})}{s_i} & \mathbf{h}_i &= \sum_j a_{ij}\mathbf{v}_j + \sum_k \tilde{a}_{ik}\tilde{\mathbf{v}}_{ik}
\end{aligned} \tag{1}
$$

Position bias. For dense attention within the local context, we use the T5 relative position bias (Raffel et al., 2020). As noted by Dai et al. (2019), adding a global position encoding to each token does not work well when processing long documents. We don't use a position bias for the retrieved memories. Experiments on the PG19 dataset (Sun et al., 2021) have shown that relative position does not appear to matter at long range, and the T5 relative bias puts all long-range tokens in the same bucket anyway.

Batching. Figure 2 illustrates how multiple long documents of different lengths are packed into a batch, and split into subsequences. Each subsequence in the batch comes from a different document, and thus requires a separate external memory, which is cleared at the start of each new document.

3.2 DISTRIBUTIONAL SHIFT

Because each long document is processed over multiple training steps, there is a distributional shift in the keys and values that are stored in external memory. The model parameters that produce the queries change over time, and will thus have shifted since the keys and values were stored. For very large memories, older records may become "stale." (Similar observations have been made for CrossBatch memory (Wang et al., 2020b) in the vision domain.) In some of our experiments, we observed that training models from scratch with a large memory sometimes resulted in worse performance than pretraining the model with a small memory of size 8192, and then finetuning it on a larger memory.

Table 3: Results on arXiv Math dataset. The models above the midline were trained from scratch, and the models below were finetuned from the 8192 memory model, written *before* $\rightarrow$ *after* as the perplexities before and after fine-tuning.

Context	Memory	Perplexity
512	None	3.725
2048	None	3.180
512	2048	3.019
512	8192	2.927
512	32k	2.877 $\rightarrow$ 2.863
512	65k	2.872 $\rightarrow$ 2.855
512	131k	**2.869** $\rightarrow$ **2.849**
512	262k	**2.869** $\rightarrow$ 2.852

Table 4: Studying the effect of more retrieval layers on the arXiv math data set. The models below the midline were finetuned from the 32k memory model, written *before* $\rightarrow$ *after* as the perplexities before and after fine-tuning.

Context	Memory	Perplexity
512	8192, layer 5	2.927
512	8192, layer $4 + 5$	2.907
512	32k, layer 5	2.942
512	32k, layer $4 + 5$	2.861
512	65k, layer $4 + 5$	2.838 $\rightarrow$ 2.838
512	131k, layer $4 + 5$	2.832 $\rightarrow$ **2.829**
512	262k, layer $4 + 5$	**2.829** $\rightarrow$ **2.829**

This training instability could be due to staleness. However, models seem to be able to cope with a limited degree of staleness (with the small memory) by adjusting their queries accordingly.

3.3 APPROXIMATE kNN

We employ *approximate* kNN search instead of exact kNN search for two reasons. First, it improves the efficiency of our model significantly, and, second, it allows us to explore the feasibility of using approximations of kNN in our model. We use a simple approximation of kNN for TPUs, which has a recall of about 90%, i.e. 90% of the true top k are returned in the approximate top k. There are various efficient approximate kNN algorithms available for CPU and GPU/TPU, for example through Faiss (Johnson et al., 2021) or ScaNN (Guo et al., 2020), which can scale into the billions.

4 EXPERIMENTS

We evaluate the effect of adding external memory on five language modeling tasks, all of which involve long-form text: English language books (PG-19), long web articles (C4), technical math papers (arXiv Math), source code (Github), and formal theorems (Isabelle). The results show significant improvements in the perplexity of the model with the addition of external memory. We experimented with various sizes of external memory, from 2048 to as high as 262k. On most of the datasets, we saw an initial sharp gain from adding a small external memory, followed by smaller but steadily increasing gains as the size of the memory was increased.

Interestingly, using even a small external memory of size 2048 provides a gain in perplexity which is almost as good as using a local context of size 2048 but no memory (e.g. Table 7). This is surprising, because the external memory is not differentiable, and is added only to one layer of the transformer, whereas increasing the context size is differentiable and affects all layers. We conclude that the lower layers of a transformer don't really need long-range context, and having a differentiable memory is not as important as one might suspect.

We further studied what the model was actually retrieving from external memory, by finding which tokens showed the biggest improvements in cross-entropy loss when the size of the memory was increased, and then examining the top-k retrieved memories for those tokens. We found that the model gained the most when looking up rare words, such as proper names, references, citations, and function names, where the first use of a name is too far away from subsequent uses to fit in the local context. This result is in keeping with the prior analysis of long-context transformers on PG19 (Sun et al., 2021), which found similar lookup patterns.

4.1 EXPERIMENTAL METHOD

We used a 6-layer decoder-only transformer with an embedding size of 1024, 16 attention heads of dimension 64, and an FFN hidden layer of size 4096. For all of our experiments, we used $k = 128$. Unless specified otherwise, we use the second-to-last layer (i.e., 5th layer) as the kNN augmented

Table 5: The table shows several examples of which tokens were retrieved during language modelling of arXiv math dataset. The model is retrieving names of the references from previous passages.

Query index	Input	Target	Surrounding context	Retrieved index	Retrieved surrounding context
20389	Mon	thus	bibitem{ ComtetMonthusYor }	2208	Brownian motion \cite{ ComtetMonthusYor }
16623	cha	kra	\cite{ chakrabarti }.	4677	∼1.2 of cite{ chakrabarti }
14747	as	d	\eqref{ asdfg } which	3365	begin{equation} \n \label{ asdfg .1}

attention layer. We also performed experiments to show the gain from having more than one kNN augmented attention layer, and for those experiments, we augmented layers 4 and 5. We used a sentence-piece (Kudo & Richardson, 2018) tokenizer with a vocabulary size of 32K, label smoothing of 0.1, and a dropout of 0.1.

We used the Adam optimizer (Kingma & Ba, 2015). In our preliminary experiments, we conducted a learning rate search among three learning rate choices: $\{1 \cdot 10^{-3}, 3 \cdot 10^{-4}, 1 \cdot 10^{-3}\}$, and found $3 \cdot 10^{-4}$ work the best. Hence we used $3 \cdot 10^{-4}$ for all the experiments. We also used a linear warmup schedule for the first 1000 steps, followed by square root decay. We adjusted the batch size so that the total number of tokens in a batch remains constant, at 65536 tokens (in the local context) per batch. We trained the models from scratch for 500K steps on all the datasets, except Isabelle. Isabelle is a small dataset, so we stopped training after 100K steps when the model began to overfit.

We ran all of our experiments on 32 TPU cores. Our models were implemented in Jax (Bradbury et al., 2018) and Flax (Heek et al., 2020). Besides the additional keys and values stored in the memory, there was only limited memory overhead. The runtime overhead on TPUs is dominated by the cost of the gather operation in kNN lookup, which is very slow on TPUs due to their memory architecture. This resulted in a slowdown of 2.4 times compared to the baseline without memory. However, the performance overhead did not depend strongly on the size of the memory. and there are highly efficient algorithms such as ScaNN (Guo et al., 2020) (CPU-based) or Faiss (Johnson et al., 2021) (GPU-based) for kNN search, and our technique can work interchangeably with any of these libraries.

We noticed that if we trained models with a large memory from scratch, the training became unstable, possibly due to distributional shift early in the training (See Section 3.2). Thus, for large memories, we first pretrain the model with a memory size of 8192 for 500K steps, and then finetune it on a much larger memory for an additional 5000 steps. We show results for the large memories both before and after fine-tuning, written *before* → *after*.

We found that increasing the size of the memory provides an immediate gain, even before any fine-tuning. This result stands in stark contrast to the usual issue with transformers, which is that a model trained using one sequence length does not generalize to longer sequences (Press et al., 2021). We hypothesize that our model is able to generalize to longer memories because the kNN lookup always returns k results, with no position bias, regardless of the size of the memory.

4.2 ARXIV MATH

For the arXiv dataset, we collected a corpus of papers by downloading them via the arXiv Bulk Data Access[1]. We filtered papers to include only articles labeled as "Mathematics" and whose LaTeX source was available. The number of tokens per paper in this dataset is roughly comparable to the number of tokens per book in PG19, because LaTeX source has many special characters and the tokenizer tends to output small subwords. Results are shown in Table 3.

Increasing the memory size. The arXiv math dataset displays the initial sharp drop in perplexity from adding external memory, and demonstrates that a memory of size 2048 is about as good as training with a local context length of 2048. Even without finetuning, increasing the size of the memory leads to a steady improvement in perplexity, from 2048 up to 131K. Note that training with external memory of size 8192, and then testing with a larger memory of 32k (Table 3) actually yields better results than training from scratch with 32k (Table 4).

The drop in perplexity from 2.927 at a memory size of 8192, to 2.877 at a memory size of 32k is significant, especially since it requires no additional finetuning, and no additional parameters have been added to the model. Nevertheless, we did observe diminishing returns as the length of the memory exceeds the typical length of papers in the dataset.

[1] https://arxiv.com/help/bulk_data

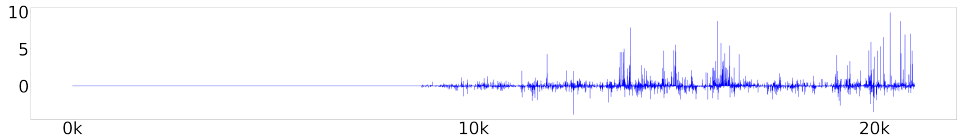

Figure 6: Difference in loss for each token in a randomly chosen paper, using the same model once with a memory size of 8k and once with 32k. Higher numbers mean the longer memory helped in comparison to the shorter memory. This paper is 22k tokens long.

Finetuning. An additional 5000 steps of finetuning further improves perplexity. Although the model can make use of additional memory without finetuning, it does benefit from being allowed to adapt to the increased memory size.

Number of kNN layers. We also experimented with using two kNN layers, rather than just one. Using two kNN attention layers does offer a modest improvement over a single layer. In general, it is better two have two smaller memory layers, than one large layer twice the size.

Where is memory most useful? Figure 6 shows a visualization of which tokens show an improvement when the size of the external memory is increased. We selected a math paper at random, and plotted the difference in cross entropy loss for each token x_i in the paper, comparing two models with the same parameters, but with memories of different sizes. $\Delta_i = \text{cross-entropy}_{8192}(x_i) - \text{cross-entropy}_{32k}(x_i)$. Positive values show an improvement in loss.

The x-axis on the chart is the token number i, while the y-axis is Δ_i. For the first 8192 tokens, the difference between the two models is zero, since the larger capacity of the 32k memory isn't being used yet. However, after token 8193, we can see that the larger memory helps, on average, over the smaller memory. The benefit is not universal, since the predictions for some tokens become worse, possibly due to the fact that a relevant retrieved memory no longer makes it into the top-k when the size of the external memory is increased. This figure also shows that the benefit of external memory is somewhat sparse. The improvement in perplexity seems to be mainly driven by a small percentage of tokens that obtain a large improvement in cross-entropy loss when using the larger memory.

How is the memory being used? Given that only a subset of tokens shows improvement from external memory, we did a further investigation into exactly what those tokens are using the memory for. We took the same math paper illustrated in Figure 6 as a case study, and filtered out those tokens which showed the largest improvement in cross-entropy loss. For each of those tokens, we examined which previous tokens appeared in the top-k retrieved memories.

Several examples are shown in Table 5, which includes both the retrieved token and its surrounding context. We observe that many of the gains in cross-entropy loss took place when trying to predict the name of bibitems, citations, or references, by looking up the references and citations used previously in the paper. Such lookups usually span over the entire paper, which is much longer than 8192 tokens, providing a plausible explanation for the gain beyond memory size of 8192.

4.3 GITHUB

We used BigQuery[2] to obtain a large corpus of Github repositories that are published with open-source licenses. We used file endings to filter for files in the languages C, C++, Java, Python (including Jupyter notebooks), Go, and TypeScript. Individual source code files are often fairly short, and there are many dependencies and cross-references between files in the repository. To capture these dependencies, we created one long document for each Github repository by traversing the directory tree, and concatenating all of the files within it. The order in which files are traversed within the repository is random, but each subdirectory is processed as a unit, so that all the files within the subdirectory are close to each other in the resulting document. Source code is usually structured so that related files are all grouped together in the same subdirectory; this traversal preserves that structure, while still shuffling files and subdirectories in random order.

Results on Github are shown in Table 7. As before, numbers below the midline are trained with memory size 8192, and then finetuned on a larger memory size, and we show perplexity both before and after finetuning. The Github results largely mirror the arXiv results. There are steady gains

[2] https://console.cloud.google.com/marketplace/product/github/github-repos

Table 7: Results on Github. The models below the midline were finetuned from the 8k memory model, written *before* → *after* as the perplexities before and after fine-tuning.

Context	Memory	Perplexity
512	None	3.111
2048	None	2.484
512	2k	2.485
512	8k	2.424
512	32k	**2.411→2.378**
512	65k	2.422→**2.376**
512	131k	2.434→2.386

Table 8: Results on Isabelle. The models below the midline were finetuned from the one-retrieval-layer 32k memory model, written *before* → *after* as the perplexities before and after fine-tuning.

Context	Memory	Perplexity
512	None	2.838
2048	None	2.326
512	2k	2.349
512	8k	2.283
512	32k	2.273
512	32k, layer $4+5$	**2.239**
512	65k	2.276→2.274
512	131k	2.280→**2.268**
512	262k	2.282→2.278

Table 9: Examples of memory retrieval in the Github dataset. The model looks up how functions are used elsewhere in the repository.

Query index	Input	Target	Surrounding context	Retrieved index	Retrieved surrounding context
23837	Fo	nte	menu_play-> setarFonte	14607	menu_load-> setarFonte
23825	,	35	hscreen/2-50, 50, 200, 35);	14599	20, y+40, 200, 35)
14546	->	adi	panel-> adicionaComponente	5205	panel-> adicionaComponente

in perplexity with increasing memory size, but this time only up to 32k. There is still a slight improvement at 65k, but only after finetuning. Interestingly enough, most Github repositories are small, with less than 65k tokens, which may explain the diminishing returns.

As with the arXiv papers, we also studied which tokens the model retrieved from memory. As might be expected, the model is often looking up the names of functions, and variables, as shown in Table 9.

4.4 Formal Math (Isabelle)

The Isabelle corpus consists of formal mathematical proofs of theories. We collected all 627 theories available on The Archive of Formal Proofs[3] (as of October 6, 2021) and an additional 57 theories from the Isabelle standard library[4] to create a corpus of 684 theories. All theories have open-source licenses. Each theory is a self-contained mathematical object, on topics such as foundational logic, advanced analysis, algebra, or cryptography, and consists of multiple files containing proofs. As with the Github corpus, all files that make up a theory are concatenated together into one long document. Unlike the Github corpus, we order the files according to their import dependencies, so that later files use sub-theorems that are proved in earlier files.

The results for Isabelle are shown in Table 8, and are similar to both Github and arXiv. Isabelle proofs can be quite long, and the results reflect that, with steadily improving perplexity up to a memory size of 131k. Adding a second kNN layer provides a greater benefit than having a larger memory.

Retrieving mathematical definitions. Our case study on the Isabelle corpus provides one of the clearest illustrations of how a model can make good use of external memory. When predicting the name of a mathematical object or a lemma, the model looked up the definition from earlier in the proof. Examples of this behavior are shown in Table 10. In example 1, the model retrieves a definition within the body of a lemma, `markov_inequality`. In example 2, it retrieves the definition of a previously defined concept `subgraph_threshold`. In example 3, it retrieves the definition of `orthonormal_system`. We manually checked 10 examples where the model made a prediction of lemma names, and 8 out of 10 times model found the body of the lemma it needs to predict. In the other two cases, the model also looked up materials in the immediate vicinity. To the best of our knowledge, this is the first demonstration that attention is capable of looking up definitions and

[3] https://www.isa-afp.org/topics.html
[4] https://isabelle.in.tum.de/

Table 10: Examples of memory retrieval in the Isabelle dataset. The model is able to find the definition of a lemma from a reference to it. The retrieved surrounding context (highlighted) is the definition body of the mathematical object highlighted in the querying context.

Query index	Input	Target	Surrounding context	Retrieved index	Retrieved surrounding context
29721	mark	ov	rule prob_space. markov_inequality	8088	M. t \<le> X a} \<le> expectation X / t"
40919	_	th	= (subgraph_threshold H n / p n)	27219	threshold H n = n powr (-(1 / max_density'
49699	S	w	assumes " orthonormal_system S w"	28050	definition orthonormal_system :: "

Table 11: Results on C4-4096+. The models below the midline were finetuned from the 8k memory model, written *before* $\rightarrow$ *after* as the perplexities before and after fine-tuning.

Context	Memory	Perplexity
512	None	21.889
2048	None	20.947
512	2k	19.375
512	8k	19.011
512	32k	**18.954**$\rightarrow$**18.916**
512	65k	**18.954** $\rightarrow$18.935

Table 12: Results on PG19. All models were trained from scratch in this table.

Context	Memory	Perplexity
512	None	13.1
2048	None	12.4
512	2k	12.06
512	8k	**11.9**
512	65k	**11.9**

function bodies from a large corpus. The Isabelle case study used the 2-layer retrieval model trained with an external memory of size 32k.

4.5 C4 AND PG-19

C4, the colossal cleaned common crawl, is a very large collection of documents that have been scraped from the internet (Raffel et al., 2020). Since many of the documents in C4 are very short, we filtered out all documents that have less than 4096 tokens. PG-19 is a large dataset of English-language books, published prior to 1919, which were retrieved from the Project Gutenberg archive (Rae et al., 2020; Sun et al., 2021). PG-19 is one of the few public datasets that only contains full-length books, and has become a benchmark for long-range natural language text modeling.

The results for C4 and PG-19 are shown in Tables 11 and 12, respectively. As before, external memory shows gains in perplexity, but only up to 32k for C4, and only up to 8k for PG-19. Most documents in C4 are less than 32k tokens long, but the same is not true of PG-19, so the diminishing returns for PG-19 are something of a surprise. Long-range references, which were common in citations (arXiv), function names (Github), and premises (Isabelle), may simply be much rarer in PG-19.

5 CONCLUSION

We presented a simple extension to the transformer architecture, called kNN-augmented attention, which dramatically increases the length of the context that a language model can attend to by using k-nearest-neighbor lookup into a large external memory. We demonstrated the effectiveness of external memory in a series of language modeling experiments over a variety of long-document datasets, including LaTeX documents, source code, formal proofs, and books.

We showed that external memory has several advantages. First, our model showed improved perplexity with larger memory size on every data set that we studied. There is a point of diminishing returns, but only at much longer context lengths than are usually used for training transformers. Second, even a very small external memory of size 2048 results in a perplexity which is comparable to using dense attention of length 2048. Third, the models we train are capable of generalizing to larger memories; a model which is trained with a small external memory of 8192 tokens can make use of a much larger memory during inference.

Unlike other forms of attention, kNN retrieval can be easily scaled up to huge memory sizes, and is thus potentially able to leverage vast knowledge bases or code repositories. How to make the best use of this capability is a topic for future work.

ETHICS

The ability to memorize large databases of facts could have potential ramifications for society, especially if those databases include sensitive personal information or copyrighted works. However, one advantage of using an external memory is that the memory can be easily cleared of all such information, as we do at the end of each document that we train on. The same is not true of differentiable model parameters, which is where most existing architectures store facts and information that they are trained on.

REPRODUCIBILITY

Details of our architecture and training hyperparameters are given in Section 4.1. The datasets for C4 and PG-19 are publicly available. Our additional datasets, Github, Isabelle, and ArXiv Math are derived from publicly available data buckets, which we link in the main part of the paper. Subsections 4.2, 4.3, and 4.4 include details on how we constructed the datasets from those datasets. We plan to release our code as open source.

REFERENCES

Joshua Ainslie, Santiago Ontañón, Chris Alberti, Vaclav Cvicek, Zachary Fisher, Philip Pham, Anirudh Ravula, Sumit Sanghai, Qifan Wang, and Li Yang. ETC: encoding long and structured inputs in transformers. In *EMNLP*, 2020.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *NeurIPS*, 2020.

Krzysztof Marcin Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamás Sarlós, Peter Hawkins, Jared Quincy Davis, Afroz Mohiuddin, Lukasz Kaiser, David Benjamin Belanger, Lucy J. Colwell, and Adrian Weller. Rethinking attention with performers. In *ICLR*, 2021.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc Viet Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In *ACL*, 2019.

Angela Fan, Thibaut Lavril, Edouard Grave, Armand Joulin, and Sainbayar Sukhbaatar. Addressing some limitations of transformers with feedback memory. *arXiv preprint arXiv:2002.09402*, 2020.

Angela Fan, Claire Gardent, Chloé Braud, and Antoine Bordes. Augmenting transformers with knn-based composite memory for dialog. *Transactions of the Association for Computational Linguistics*, 9:82–99, 2021.

Edouard Grave, Armand Joulin, and Nicolas Usunier. Improving neural language models with a continuous cache. In *ICLR*, 2017.

Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *ICML*, 2020.

Ankit Gupta, Guy Dar, Shaya Goodman, David Ciprut, and Jonathan Berant. Memory-efficient transformers via top-k attention. *CoRR*, abs/2106.06899, 2021. URL `https://arxiv.org/abs/2106.06899`.

Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. Retrieval augmented language model pre-training. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pp. 3929–3938. PMLR, 2020. URL http://proceedings.mlr.press/v119/guu20a.html.

Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. Flax: A neural network library and ecosystem for JAX, 2020. URL http://github.com/google/flax.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 2021.

Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Generalization through memorization: Nearest neighbor language models. In *ICLR*, 2020.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *EMNLP*, 2018.

Guillaume Lample, Alexandre Sablayrolles, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. Large memory layers with product keys. In *NeurIPS*, 2019.

Mike Lewis, Marjan Ghazvininejad, Gargi Ghosh, Armen Aghajanyan, Sida Wang, and Luke Zettlemoyer. Pre-training via paraphrasing. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 18470–18481. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/d6f1dd034aabde7657e6680444ceff62-Paper.pdf.

Ofir Press, Noah A. Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. *CoRR*, arXiv preprint abs/2108.12409, 2021.

Jack W Rae, Anna Potapenko, Siddhant M Jayakumar, Chloe Hillier, and Timothy P Lillicrap. Compressive transformers for long-range sequence modelling. In *ICLR*, 2020.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 2020.

Hongyu Ren, Hanjun Dai, Zihang Dai, Mengjiao Yang, Jure Leskovec, Dale Schuurmans, and Bo Dai. Combiner: Full attention transformer with sparse computation cost. *arXiv preprint arXiv:2107.05768*, 2021.

Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. *Transactions of the Association for Computational Linguistics*, 9:53–68, 2021.

Sainbayar Sukhbaatar, Edouard Grave, Guillaume Lample, Herve Jegou, and Armand Joulin. Augmenting self-attention with persistent memory. *arXiv preprint arXiv:1907.01470*, 2019.

Sainbayar Sukhbaatar, Da Ju, Spencer Poff, Stephen Roller, Arthur Szlam, Jason Weston, and Angela Fan. Not all memories are created equal: Learning to forget by expiring. In *ICML*, 2021.

Simeng Sun, Kalpesh Krishna, Andrew Mattarella-Micke, and Mohit Iyyer. Do long-range language models actually use long-range context? *CoRR – arXiv:2109.09115*, 2021.

Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. Long range arena: A benchmark for efficient transformers. *arXiv preprint arXiv:2011.04006*, 2020a.

Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *arXiv preprint arXiv:2009.06732*, 2020b.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020a.

Xun Wang, Haozhi Zhang, Weilin Huang, and Matthew R. Scott. Cross-batch memory for embedding learning. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pp. 6387–6396. Computer Vision Foundation / IEEE, 2020b. doi: 10.1109/CVPR42600.2020.00642. URL https://openaccess.thecvf.com/content_CVPR_2020/html/Wang_Cross-Batch_Memory_for_Embedding_Learning_CVPR_2020_paper.html.

Ronald J. Williams and Jing Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation*, 1990.

Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontañón, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: Transformers for longer sequences. In *NeurIPS*, 2020.

Zhenhai Zhu and Radu Soricut. H-transformer-1d: Fast one-dimensional hierarchical attention for sequences. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli (eds.), *ACL*, 2021.

A LENGTH OF INPUTS

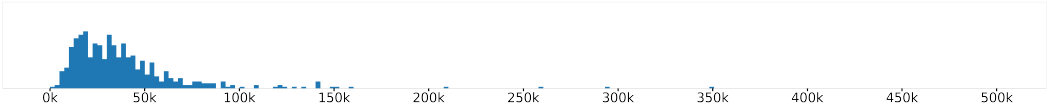

Figure 13: Histogram of the number of tokens in Arxiv Math papers dataset. We tuncated the histogram at 500k tokens. The maximum paper had almost 1.6M tokens.

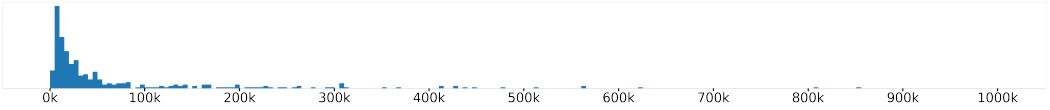

Figure 14: Histogram of the number of tokens in Github repositories dataset. We cut off the long tail of this plot. The repository with the maximum length has just over 9M tokens.

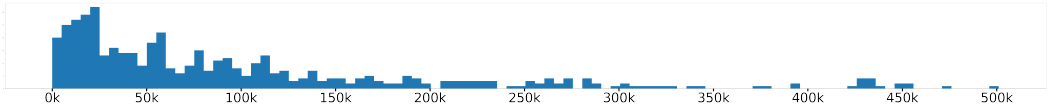

Figure 15: Histogram of the number of tokens in Isabelle proof scripts dataset.

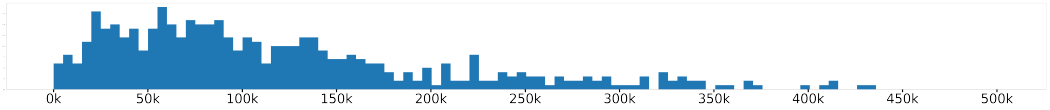

Figure 16: Histogram of the number of tokens in PG19 books dataset.

Figure 17: Histogram of the number of tokens in C4 documents filtered by documents that have less than 4096 tokens.

B MORE RETRIEVING EXAMPLES IN FORMAL THEOREM PROVING CORPUS

Example 1

- Input token index: 64604
- Input token: "_"
- Target token: "pair"
- Surrounding context:)) by (simp add: Fourier_sum_limit_pair [OF f, symmetric] Fourier'
- Name needs to be predicted: `Fourier_sum_limit_pair`
- Retrieved token: "Four"
- Retrieved token index: 64412
- Retrieved context: 2 * n. Fourier_coefficient f k * trigonometric_set k t)
- Definition of the name:

```
lemma Fourier_sum_limit_pair:
  assumes "f absolutely_integrable_on {-pi..pi}"
  shows "(λn. ∑k≤2 * n. Fourier_coefficient f k * trigonometric_set k t) ⟶ l
     ⟷ (λn. ∑k≤n. Fourier_coefficient f k * trigonometric_set k t) ⟶ l"
       (is "?lhs = ?rhs")
```

Figure 18: Definition of `Fourier_sum_limit_pair`.

Example 2

- Input token index: 46175
- Input token: "tri'"
- Target token: "gon"
- Surrounding context: <le>n. a k * trigonometric_set k x)
- Name needs to be predicted: `orthonormal_system_trigonometric_set`
- Retrieved token: "gon"
- Retrieved token index: 35457
- Retrieved context: lemma orthonormal_system_trigonometric_set:\n "orthonormal_system
- Definition of the name:

```
lemma orthonormal_system_trigonometric_set:
    "orthonormal_system {-pi..pi} trigonometric_set"
```

Figure 19: Definition of `orthonormal_system_trigonometric_set`.

Example 3

- Input token index: 49760
- Input token: “sum”’
- Target token: “m”
- Surrounding context: nusing Fourier_series_square_summable [OF assms, of’
- Name needs to be predicted: `Fourier_series_square_summable`
- Retrieved token: “sum”
- Retrieved token index: 35457
- Retrieved context: lemma Fourier_series_square_summable\n assumes:
- Definition of the name:

```
lemma Fourier_series_square_summable:
  assumes os: "orthonormal_system S w" and w: "⋀i. (w i) square_integrable S"
    and f: "f square_integrable S"
  shows "summable (confine (λi. (orthonormal_coeff S w f i) ^ 2) I)"
```

Figure 20: Definition of `Fourier_series_square_summable`.

Example 4

- Input token index: 49697
- Input token: “_”’
- Target token: “system”
- Surrounding context: lemma Riemann_lebesgue_square_integrable:
  nassumes "orthonormal_system S w
- Name needs to be predicted: `orthonormal_system`
- Retrieved token: “system”
- Retrieved token index: 28052
- Retrieved context: definition orthonormal_system :: "\’a::euclidean’
- Definition of the name:

```
definition orthonormal_system :: "'a::euclidean_space set ⇒ ('b ⇒ 'a ⇒ real) ⇒ bool"
  where "orthonormal_system S w ≡ ∀m n. l2product S (w m) (w n) = (if m = n then 1 else 0)"
```

Figure 21: Definition of `orthonormal_system`.

Example 5

- Input token index: 34817
- Input token: “.”’
- Target token: “b”
- Surrounding context: shows "integrable (lebesgue_on {a..b})
- Retrieved token 1: “.”
- Retrieved token index 1: 2416
- Retrieved context 1: lebesgue_on {a..b}) f i
- Retrieved token 2: “-”
- Retrieved token index 2: 2445
- Retrieved context 2: (lebesgue_on {a-c..b-c}) (
- Retrieved token 3: “-”
- Retreived token index 3: 6479
- Retrieved context 3: (lebesgue_on {-pi..pi}) (

Example 6

- Input token index: 49759
- Input token: “_”’
- Target token: “sum”
- Surrounding context: 0"\n using Fourier_series_square_summable [OF assms
- Retrieved token 1: “set”
- Retrieved token index 1: 35044
- Retrieved context 1: definition trigonometric_set :: "nat \<Rightarrow>
- Retrieved token 2: “ier”
- Retrieved token index 2: 47272
- Retrieved context 2: definition Fourier_coefficient\nwhere
- Retrieved token 3: “ine”
- Retrieved token index 3: 18160
- Retrieved context 3: lemma Schwartz_inequality_strong:\nassumes “f’
- Retrieved token 4: “system”
- Retrieved token index 4: 28052
- Retrieved context 4: definition orthonormal_system :: “\’a::euclidean’
- Retrieved token 5: “<”
- Retrieved token index 5: 47241
- Retrieved context 5: subsection\<open>Convergence wrt the L’
- Retrieved token 6: “n”
- Retrieved token index 6: 40835
- Retrieved context 6: \n subsection\<open>A bit of extra’