Implementing A Neural Cache LSTM

Christina Wadsworth	Raphael Palefsky-Smith
Department of Computer Science	Symbolic Systems Program
Stanford University	Stanford University
Stanford, CA 94305	Stanford, CA 94305
cwads@cs.stanford.edu	*rpalefsk@stanford.edu*

Abstract

We re-implement the Grave et al. Neural Cache on our own LSTM model, reproducing the perplexity results and performing additional hyper parameter tuning. We additionally test the perplexity of different authors' text on other authors' trained LSTMs with the Neural Cache implementation.

1 Introduction

Language models are a vital tool for Natural Language Processing, underlying many higher-level applications like question answering and machine translation. The task is simple: given a sequence of words, compute the conditional probability of the next word. This can be well approximated by neural networks, especially LSTM models that can capture long-term dependencies. In the quest to improve these models' performance, one approach is to increase the size of the LSTM hidden layer, or stack multiple layers to give the model more "memory." While effective, this comes at the cost of many more parameters, and therefore the need for longer training times and more data.

One alternative is memory-augmented networks. These systems give the network access to external, non-parameterized memory at test-time. These allow the network to remember more context and improve performance without the burden of additional parameters. An especially simple, and remarkably performant, memory-augmented network architecture is Grave et al.'s Continuous Cache [1]. This module has several advantages: it is straightforward and fast, runs entirely at test-time with no training required, and can be attached atop any Recurrent Neural Network without modifying the underlying architecture. Furthermore, it can predict Out-of-Vocabulary words after encountering them just once in the test input and storing them in the cache. And most importantly, it consistently reduces language models' perplexity on standard benchmarks.

Our project consists of a reimplementation of Grave et al.'s work and a comparison on the Wikitext2 perplexity benchmark. We perform extensive hyper parameter optimization, conducting a grid search over the cache's *alpha* and *theta* parameters to tune its performance. Finally, we apply this optimized model to literary analysis, fine-tuning the Wikitext2 LSTM model on works by Charles Dickens, Mark Twain, and H.G. Wells and examining each model's perplexity on the other authors' work, aiming to identify hot spots and determine if any semantic or structural characteristics are consistent between authors.

2 Background/Related work

The Neural Cache model draws from two main inspirations: memory-augmented networks and cache models. Memory-augmented networks – the most prominent of which is DeepMind's Neural Turing Machine [2] – learn to read and write from an external memory store. These read and write operations are fully differentiable, so the use of memory is optimized like any other part of the network, via gradient descent. Memory-augmented networks are able to store much more information than un-augmented networks, boosting their performance on context-sensitive tasks like language modeling. However, according to Grave et al., these networks are computationally expensive, and this overhead limits the models' practical memory capacity. So, Grave et al seek a more lightweight approach, one that can store information like a memory-augmented network but without the computational cost.

In this vein, Grave et al. re-introduce the concept of a *cache*. First implemented by Kuhn and De Mori [3] in 1990, language model caches store a window of previously encountered words. Intuitively, if a word appears once, it is more likely to appear again. For instance, a recipe containing flour is likely to repeat the word "flour" many times. Cache models take advantage of this property and assign higher prediction probabilities to words already stored in the cache. These modules are fast, require no training, and unlike memory-augmented network architectures, can be grafted onto existing models without modification.

Grave et al.'s Neural Cache can be considered a synthesis of these two ideas. Much like Kuhn and De Mori's work, the Neural Cache is a simple cache tacked onto the top of an already-trained model. But unlike Kuhn and De Mori's cache, which weights all cached words equally, the Neural Cache weights each word by its *hidden state similarity*. When each word is added to the cache at runtime, it is associated with the LSTM hidden state that produced it. To predict the next word, the text is first run through the unmodified neural network. Then, the hidden state of this network is input to the cache. This state is dotted with each hidden state in the cache, and the associated words' probabilities are weighted by this product (and *theta* and *alpha* hyper parameters). The hidden state weighting acts much like a memory-augmented network by linking memory access to the internal state of the network. But there is none of the computational overhead, as the memory read/write operations need not be learned. In a sense, the Neural Cache is the best of both worlds: the power of memory augmentation with the speed of a cache.

3 Approach

Our experiments are divided into two phases. In the first phase, we train an LSTM language model on the Wikitext2 corpus, with additional fine-tuned models trained on works by Charles Dickens, Mark Twain, and H.G. Wells. In the second phase, we apply the cache at test-time, feeding the test set predictions of an already-trained model through our cache implementation.

Our language model is a 1024-unit LSTM implemented with Keras. It consists of a fully-connected embedding layer transforming the vocabulary size to the 1024-length hidden state size, a single LSTM layer, and a fully-connected output layer transforming the LSTM's hidden state back to a vocabulary-sized logits vector. Finally, the logits are passed through a softmax function to compute a probability distribution for the next word.

Our sequence length (the number of unrollings through time) is 30, and our batch size is 20. We apply a categorical cross-entropy loss at every step in time: at each step, the network is trained to predict the next word in the sequence. We use the ADAM optimizer with a learning rate of 1e-3 and a per-epoch weight decay of 2e-5 over 50 epochs.

At test-time, we run the network on the entire test set and record – for each word - the softmax output, the LSTM's hidden state, and the raw logits. The softmax output is used to benchmark the baseline, un-cached model, and the logits and hidden state are fed into our cache implementation.

Our cache implementation integrates the cache probability below into the probability distribution of the vocabulary:

$$p_{cache}(w \mid h_{1..t},\ x_{1..t}) \propto \sum_{i=1}^{t-1} \mathbb{1}_{\{w=x_{i+1}\}} \exp(\theta h_t^\top h_i)$$

If word *w* is in the cache, the similarity product between the current hidden state and the hidden state stored in the cache with word *w* is calculated and multiplied by hyper parameter *theta*. The idea here is that if a word has been seen previously as the "true" output word of a hidden state and that hidden state is similar to our current hidden state, the word *w* is more likely to be the next output word. Below, the cache probability is factored into the probability distribution:

$$p(w \mid h_{1..t},\ x_{1..t}) \propto \left(\exp(h_t^\top o_w) + \sum_{i=1}^{t-1} \mathbb{1}_{\{w=x_{i+1}\}} \exp(\theta h_t^\top h_i + \alpha) \right)$$

The above equation is referred to as global normalization, and represents a softmax over the vocabulary and the words in the cache. In another formulation, the vocabulary and cache probability are linearly interpolated with a *lambda* parameter as follows:

$$p(w \mid h_{1..t},\ x_{1..t}) = (1-\lambda) p_{vocab}(w \mid h_t) + \lambda p_{cache}(w \mid h_{1..t}, x_{1..t})$$

We focused on the global normalization probability distribution, computing the probability for each word, then taking the softmax over the vocabulary.

3.1 Global normalization, with vectorization

Calculating perplexity requires only the model's probability estimate of the true class. Given the large vocabularies involved – Wikitext2 contains over 33k words – it is significantly faster to compute a probability for a single word rather than the entire vocabulary. So, we exploit this property to achieve a computational speedup. We vectorized the cached hidden states and the output weights to calculate the sum over the vocabulary and the entire cache, which is the denominator of the softmax equation, with only two matrix multiplications. We are able to use this exploitation because when we sum the denominator by word, we search the cache to find all pairs containing that word and use that pair's corresponding hidden state. Because we are summing over all words, we will search for each word once, and therefore we will retrieve each cache entry once. Therefore, we can circumvent this individual search by simply taking all hidden states in the cache and vectorizing them to be multiplied by the current hidden state. This decreased runtime over our model that found each cache probability individually and summed those individual probabilities.

3.2 Global normalization, without vectorization

Our vectorization approach is useful for faster perplexity calculations, but for applications such as text generation, we need the probability estimates for every word in the vocabulary. First, we initialize this vector to the neural network's probability estimate for each word. Then, we simply loop over each word in the cache, computing a cache probability for that word and adding it to the corresponding row in the initialized vector. While this method is slower, it lets us generate text and serves as a vital sanity check for the computational shortcut described in Section 3.1.

4 Experiments

The experiments we performed emulated Grave et al.'s experiments. The most important benchmark and performance to test was perplexity. We chose the Wikitext2 dataset to test on. Of all the datasets Grave et. al. tested on, the Wikitext2 data set was the smallest, and therefore the easiest for us to reproduce tests on. The different computational techniques described in Section 3 produced identical results, but for the sake of computation time, we ran the tests using the Section 3.1 "shortcut" technique with vectorized matrices.

4.1 Perplexity

Model	**Testing**
Neural cache model (size = 100) (Grave et. Al 2016)	81.6
Neural cache model (size = 2000) (Grave et. Al 2016)	68.9
Neural cache model (size = 100) (Our model)	82.2
Neural cache model (size = 500) (Our model)	69.9
Neural cache model (size = 2000) (Our model)	64.7

Table 1: Best perplexity results on Wikitext-2

We saw similar numbers to the Grave et. al. paper. Our perplexities are within a few points of theirs, and differences can be explained by the difference in the base models. Our base LSTM models are separate implementations (different weight initialization, learning rate, and non-adaptive softmax function) so the perplexities should not be identical. However, our model follows the same trends, which can be seen more clearly below.

4.2 Cache sizes

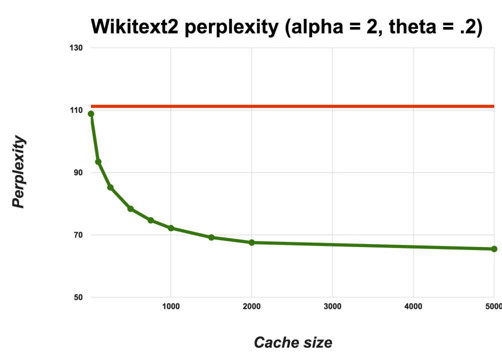


Figure 1: Perplexity graphed with different cache sizes using the Wikitext2 test set, alpha and theta set at 2 and .2, respectively.

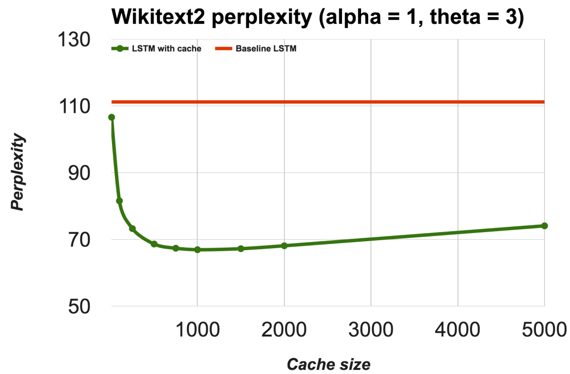


Figure 2: Perplexity graphed with different cache sizes using the Wikitext2 test set, alpha and theta set at 1 and 3, respectively

In the second graph, we selected hyper parameters *alpha* and *theta* that are most optimal for the size 500 cache. We weight the cache too much at 5000, so we actually see a rise in perplexity after a cache size of about 1500. A smaller theta is better for our larger cache sizes, as we can see in the first graph. Our first graph shows hyper parameters tuned to a larger cache of size 2000. We see a less steep decline in the beginning, but a decline still to size 5000.

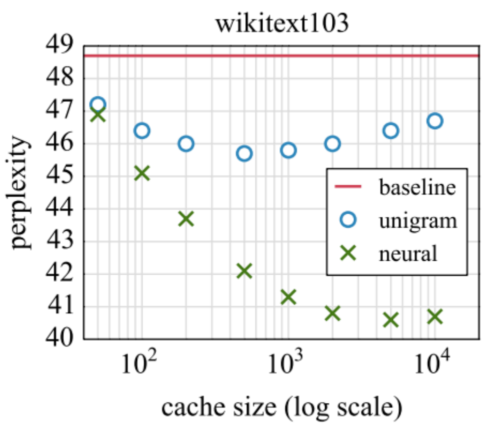
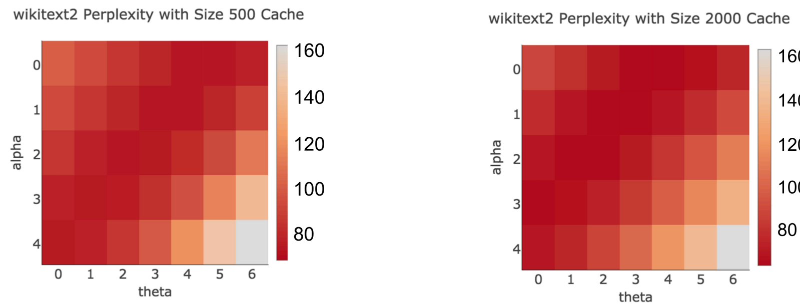


Figure 3: The Grave et al. Neural Cache's perplexity on Wikitext103 graphed with different cache size

As we see from the above graph of perplexity calculated on wikitext103 by the Grave et al. Neural Cache, our implementation has the same trend as the Grave et al. cache. An important disclaimer here is that the data sets are different, but Grave et al. did not include a wikitext2 graph and the graph above is still useful as a comparison. We see the divergence from the baseline follow a similar trend, and we even see the uptick at the end when the cache size surpasses the optimal hyper parameters. Our graph with hyper parameters *alpha = 3, theta = 3* follows the same trends.

4.3 Hyper parameter tuning

Hyper parameter *alpha* weights the Neural Cache. Hyper parameter *theta* weights the similarity product within the cache. Below, we show two graphs: one with perplexities calculated using a cache of size 500 and one with a cache of 2000. As we can see, our optimal hyper parameters decrease when cache size increases. We additionally found that our most optimal *theta* was in a range approximately one order of magnitude above the optimal *theta* on the Grave et al. Neural Cache. Our underlying model was different, so this difference makes sense. Unfortunately, it is somewhat hard to compare the graphs we produced below upon first glance with the Grave et al. graph because our scaling is so much more extreme than theirs is. In general, however, our hyper parameter graphs follow the same trend that a larger *theta* between 2-4 (for Grave et al., .15-.3) and an alpha ranging from 0-2 produce the best perplexity results for the model.



Figures 4 and 5: Our hyper parameter optimization results

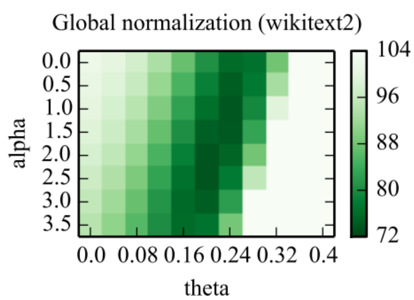


Figure 6: Grave et al.'s hyper parameter optimization results

Another thing to note is that our alpha has a slightly bigger impact than the paper's *alpha*. We were unsure about why this was the case, but we supposed that since our base models are not identical, different parts of the cache would weight differently since the hidden state is dependent on the original model. Our base model was slightly worse, perplexity-wise, than Grave et al's, so the cache as a component having a larger weight with a higher optimal *alpha* value than they found makes sense.

4.4 Results

We are confident in our reproduction. The downward trend on our perplexity graphs is extremely similar to the paper's cache size-vs-perplexity trends. We also observed differing optimal hyper parameters for different cache sizes, which suggests that cache size materially affects performance and must be tuned as part of a larger system. Given that this section of our work is a re-implementation of an existing paper, there is not much to report other than the success of our implementation. The perplexity numbers match up (within a reasonable margin owing to differing base models), and we are satisfied that our Neural Cache implementation is sound. With the Neural Cache in our toolkit, we turned to more lighthearted literary applications.

5 Literary applications

Our primary application for the Neural Cache LSTM model was evaluating *author similarity*. We trained separate models for several authors, and used these models' perplexity on the other authors' work as a proxy for similarity. Intuitively, if a language model of Author A has a low perplexity on Author B, then A and B must have relatively similar styles and word choice. At a finer level, we evaluate perplexity on length-30 subsequences, and can thereby determine which sequences match, or do not match, a particular author's style. Note that this did not specifically require the Neural Cache – any language model would have done the job – but we wanted to take advantage of the Neural Cache model's superior performance.

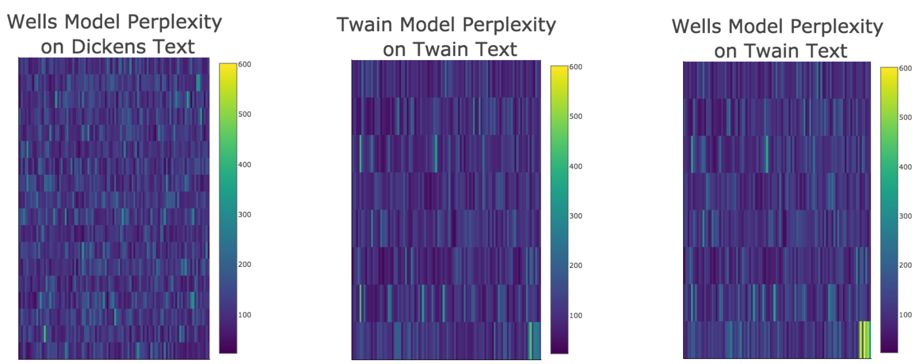
4.1 Training author models

We created corpora for H.G. Wells, Mark Twain, and Charles Dickens by concatenating their novels from the University of Michigan's cleaned subset of Project Gutenberg [4]. The works were concatenated in lexicographic order by title, and then split into train (80%), validation (10%), and test (10%) sets. We then fine-tuned our Wikitext2 model on each author's corpus. Any author words not in the Wikitext2 dataset were converted to the <unk> token. After experimenting with text generation, we were disappointed to find that the sentences produced were largely unintelligible, and not obviously discernable between authors. However, given that our focus was on similarity metrics and not text synthesis, we pressed onward.

.

4.2 Perplexity heat maps

After fine-tuning each author LSTM, we ran each model on the other authors' test sets, generating a heat map of perplexity on 30-word sequences within the text. We expected to find "hot spots" - certain paragraphs or sections that were particularly similar or dissimilar between authors.



In fact – and disappointingly for us - the majority of the heat maps have no discernable pattern, regardless of author model or input corpus. There may be a few reasons for this. First, the model trained on Wikitext2 could be dominant since our author data sets were relatively much smaller. The authors we chose were also fairly similar. We stayed away from Shakespeare since his style is such a drastic change from the above authors, but in retrospect that would have perhaps been a more interesting comparison. Additionally, a lot of tokens in the author texts were set to <unk>. However, we noticed a high-perplexity spike at the end of Twain heat maps across all author models. After manually inspecting the Twain input, we discovered that the final novel in the corpus had a different newline structure than the other sections. To investigate the effects of new lines on our perplexity, we re-ran our analysis after removing all newlines. Prior to new line removal, the Twain model on the Twain corpus exhibited **55.2 perplexity**. After removing newlines, perplexity spiked to **81.9**. Our model seems to have latched onto the easiest feature to train on and weighted it more than other, more nuanced semantic and structural differences unique to each author, causing the differences between the author-to-author heat maps to be small. Care must be taken to ensure that input is uniformly formatted and conclusions are not made before digging into data.

6 Conclusion

As neural network language models continue to improve, it is likely that memory will continue to play a larger and larger role. But as the Neural Cache has shown, these memory extensions need not be complicated or computationally intensive. They can be simple, fast, and adaptable to existing models. We successfully implemented the Neural Cache and confirmed its impressive performance benefits.

Our literary experiments did not yield any especially novel (pun intended) results. However, they served as an important reminder that LSTMs (even with the Neural Cache implemented) tend to grab onto the “easiest” set of features, in this case, the <eos> tokens. It’s important to go through data sets to see how text is formatted before drawing extrapolations about structural or semantic significance

In summary, we successfully implemented a brand-new, high-performance extension to an LSTM language model. We are excited to see the Neural Cache – and future memory-based developments – improve the state of NLP.

6.1 Contributions

Both team members contributed equally to this paper and project. Raphie wrote the initial LSTM from scratch, and Christina added the first Neural Cache implementation. Raphie then added another Cache implementation to help test and validate the first one. Both spent a significant amount of time validating the cache and its accuracy. The rest of the cache and literary graphs and tests were then run by both Christina and Raphie together, and both worked on the poster and write up together.

Acknowledgments

We are incredibly grateful to Richard Socher, who pointed us toward this project and helped us through its ups and downs. We further wish to acknowledge the authors of the Neural Cache paper, Edouard Grave, Armand Joulin, and Nicolas Usunier, both for their contribution to the field and the thoroughness of their documentation. Finally, a huge thank-you to Microsoft Azure for their generous donation of GPU time.

References

[1] Grave, Edouard, Armand Joulin, and Nicolas Usunier. "Improving Neural Language Models with a Continuous Cache." *arXiv preprint arXiv:1612.04426*(2016).

[2] Graves, Alex, Greg Wayne, and Ivo Danihelka. "Neural turing machines." *arXiv preprint arXiv:1410.5401* (2014).

[3] Kuhn, Roland, and Renato De Mori. "A cache-based natural language model for speech recognition." *IEEE transactions on pattern analysis and machine intelligence* 12.6 (1990): 570-583.

[4] Lahiri, Shibamouli. "Complexity of word collocation networks: a preliminary structural analysis." *arXiv preprint arXiv:1310.5111* (2013).