Cache Attention for Recurrent Language Modeling

Colin Gaffney
Department of Computer Science
Stanford University
cgaffney@stanford.edu

Abstract

Language modeling is likely the most fundamental task of deep natural language processing, and it has numerous practical applications from facilitating text generation by humans to helping researchers evaluate the progress and "intelligence" of AI. As a result, language modeling is a fertile ground for experimenting with new architectures and variations on existing ones. In this paper we expand on the idea of incorporating long-term memory into standard recurrent neural networks by including a cache that stores the hidden states of previous training examples. Predictions incorporating cached information can be made using an additional attention mechanism that operates over the cache. We conclude that cache attention does indeed improve the accuracy of standard attention-based recurrent language models by about 5 percentage points. We finally address inefficiencies and potential improvements to our cache implementation.

1 Introduction

Sentence completion is perhaps one of the most fundamental tasks of artificial intelligence-driven natural language processing. Its obvious applications are already included in ubiquitous products, such as iMessage and Google Search. In some contexts, however, sentence completion is not a trivial task, as it requires considerable of both syntactic and semantic knowledge, as well as a general "real-world" knowledge. As an example, consider the sentence, "Aizac is a town and commune of the Ardche dpartement, in the southern part of _." The correct response is "France," but this requires the knowledge that Aizac and Ardche belong in France, and not in Belgium, for example. At the very least, it requires an understanding that Aizac and Ardche are "French-sounding" words.

Our aim in this work is to improve on the baseline recurrent neural network, which has shown considerable effectiveness for natural language applications [1]. We will do this by adding two attention mechanisms, the first of which is a self-attentive mechanism following Vaswani et al. [2]. The second is an attention mechanism that operates over a cache of a constant number of previously seen training examples. While the effectiveness of attention has been well established, we will seek to evaluate the performance of cache attention in this work.

Our motivation for exploring cache attention is the observation that in attempting to complete the last word in a sentence, the necessary information is often not included in the sentence itself. To make accurate predictions, a model must not only be trained on a corpus of broad human knowledge (such as Wikipedia), but it must also store relevant facts about the world. Given a limited volume of memory space, we can of course only store the most relevant pieces of information. Thus, we believe that by storing information about important sentences the model has previously trained on, we may enhance future prediction accuracy.

2 Related Work

There has been considerable research into neural sentence completion, although not all of it focuses on predicting the last word. A traditional approach uses N-gram modeling [3] but this approach suffers from a limited context and sparse data for larger contexts. More recently, language modeling has been approached using neural networks, specifically, the large family of recurrent neural networks (RNN's) [4]. Mikolov et al. [5] made one of the most significant breakthroughs in this field, although in their case language modeling served as merely an objective task in learning word vector representations. The basic recurrent model has been improved significantly by additions such as long short-term memory (LSTM) cells and gated recurrent units (GRU) [6] and the attention mechanism [2]. For the task of sentence completion, various classes of models may even be merged; for example, by incorporating the output of a neural dependency parse [7].

Besides using word prediction as a task for generating word vectors, sentence completion also has many extrinsic applications. Zweig et al. [4] for example, apply sentence completion to fill-in-the-blank SAT vocabulary questions.

Language models that include some form of long-term memory have similarly been introduced in various forms. The most basic cache model simply saves a unigram representation of previous examples and uses them to predict the current example [8]. A more modern extension of this principle is proposed by Grave et al. [9], who store hidden states and word predictions as tuples in a dictionary-like structure. These states are then used to define a probability distribution for retrieving a stored word from memory.

More complex architectures include Memory Networks [10] and their variants such as Dynamic Memory Networks [11]. The former proposes a general method for inserting each input into an external memory, and prediction can be conducted by using a subset of the memory as input for a neural network. The latter trains distinct RNN's for four separate modules - the input, question, episodic memory, and answer. This architecture proposes a solution to the difficulties experienced by a single RNN on documents with a large number of timesteps. However, given our limited resources, the computational burden of training such a network is impractical, particularly since the episodic memory module requires multiple iterations to update its state. Additionally, this model differs from the approach we will detail below in that assumes the input information is structured in a long list of sentences.

3 Approach

3.1 Neural Network

Our basic approach uses a recurrent neural network to predict the last word in a sentence. We omit the prediction of the successor word at every timestep for simplicity. Formally, given a sentence of length n, our input is the vector $[x_1, ..., x_{n-1}]$ and we aim to predict x_n. Here, each x_t is a word embedding. At each timestep, a hidden state h_t is computed given the current word x_t and the previous hidden state h_{t-1}. In our specific implementation, h_t is computed using a standard LSTM cell. Finally, the last hidden state h_{n-1} is transformed into a $|V|$-dimensional vector over which a softmax is taken.

3.2 Attention

We also incorporate an attention mechanism that uses a nonlinear transformation of the final hidden state with all previous hidden states. In other words, h_{n-1} is compared against every other $h_{t \leq t-1}$ to generate an attention weight for each timestep. In this implementation we use additive attention. For a timestep t, the attention weight is given by

$$a_t = v^T tanh(Wh_t)$$

where tanh is applied element-wise. The weights are normalized into a probability distribution using a softmax function. The last hidden state h_{n-1} is then updated incorporating attention by setting

$$h_{t-1} = \sum_{t=1}^{n-1} a_t h_t$$

The word prediction may then proceed as described above. Although attention was developed for sequence-to-sequence learning to counter the bottleneck problem in extrapolating an entire sentence from a single hidden state, attention is also useful in our case assist the RNN in maintaining a longer memory. Although LSTM cells help mitigate a traditional RNN's bias towards recent inputs, attention helps the model consider all past states more equally.

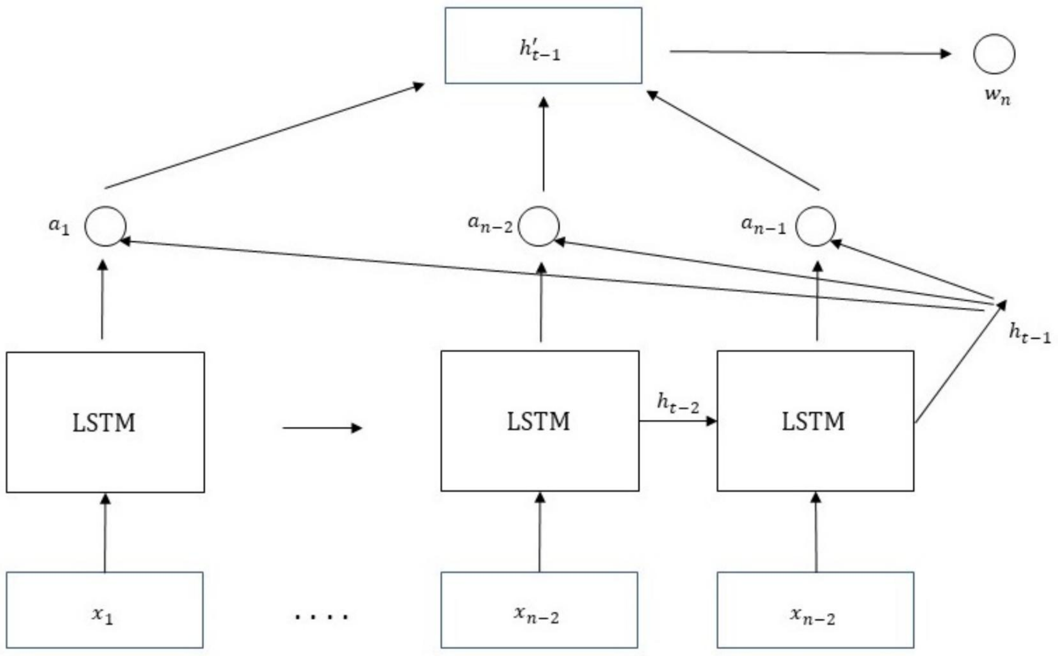

Figure 1: An LSTM RNN incorporating attention as described in (3.1) and (3.2)

3.3 External Memory

The external cache used by our model can be conceptualized in two different ways depending on whether the memory is being written to or read from.

At prediction time, when the model is reading from the cache, the cache can be viewed as a matrix C. Each row contains the last hidden state h_{n-1} of some previous training example. Hidden states stored in the cache incorporate the attention mechanism detailed above. At prediction time, the attention weights for each cache entry are computed using the same attention mechanism described above on a sentence level, where

$$a_i = v_c^T tanh(W_c C_i)$$

for a cache entry i. Whereas the sentence-level attention weights can be conceptualized as lending importance to particular words, the cache-level weights lend importance to particular entries in the matrix. The attention weighted cache is then multiplied by another weight variable and added to the final hidden state of the sentence currently being predicted.

After a prediction is made, we must now decide whether to store the final hidden state of the current input sentence or discard it. As described above, the cache attention score for a particular hidden state is easily interpretable as an "importance" metric for the sentence in question, and thus may be used to store only the highest-scoring hidden states in the cache.

At write-time, the model's memory may be conceptualized as a binary min-heap. In this construction, each node, corresponding to a hidden state, must have a lower cache attention weight than either of its children. After the model computes a hidden state for the current training example, the state will be discarded if its score is less than the score of the root. Otherwise, the current state will replace the root and then sift-down until reaching its proper location. Maintaining the cache in this manner allows us to approximate a $O(logn)$ runtime for insertions, assuming the cache has size $O(n)$. This also assumes the hidden state size, h, is much smaller than n, otherwise the insertion time is $O(hlogn)$.

Unfortunately, since W_c and v_c are variables subject to continual update, the min-ordering properties and runtime cannot be guaranteed, but merely approximated. This limitation may also inhibit the model from learning appropriate weights for each cache entry; we expect the weights to increase for the lower branches of the heap, but if the heap is not well-ordered, the weights may be uniform instead.

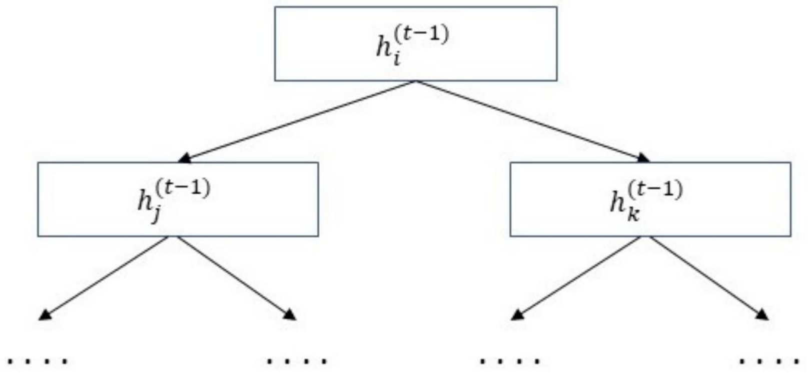

Figure 2: A binary min-heap storing hidden states from previous training examples. We hope to approximate the property that $v_c^T tanh(W_c h_i^{(t-1)})$ is less than $v_c^T tanh(W_c h_j^{(t-1)})$ and $v_c^T tanh(W_c h_k^{(t-1)})$.

4 Experiments

4.1 Data and Implementation

Our data is taken from Zhu et al. [12] which use the data for a sentence simplification task. We originally intended to perform experiments with increasing the verbosity of sentences, for which a parallel sentence corpus was necessary, but we later switched to our current task. We selected the subset of the corpus that was drawn from Wikipedia (the other half of the corpus was drawn from the Simple English Wikipedia.) We trained on a random subset of 70% of the data, evaluated on 10%, and tested on the remaining 20%.

The project was implemented using Tensorflow. All experiments were performed using a single GPU on the Microsoft Azure platform.

4.2 Evaluation Methods

Cross-entropy loss was chosen as an optimization metric for the model. For an individual training example i, the loss is computed as

$$L^{(i)} = -\sum_{j=1}^{|V|} y_j^{(i)} log(\hat{y}_j^{(i)})$$

where $(\hat{y})^{(i)}$ is the predicted label and $y^{(i)}$ is a one-hot encoding of the true label. $|V|$ denotes the size of the vocabulary.

Aside from the optimization function, our primary evaluation metric is accuracy, even though this has many flaws. For example, consider the sentence, ”In general relativity, gravitational radiation is made at times where the curvature of spacetime is moving in waves, such as is the case with co-orbiting _.” The correct answer from Wikipedia is ”objects,” but it is obvious that some other responses, such as ”bodies,” ”planets,” or ”stars” are equally plausible. If either a machine or a human responds inaccurately, it is because accuracy is an unfair metric. However, on average, we expect accuracy to capture the relative skills of humans and machines in performing the modeling task.

4.3 Results

First, to center expectations for our results, we evaluated human performance on the same task. We randomly selected a subset of 100 sentences from the entire corpus, removed the last word, and asked a group of 8 Stanford students to predict the missing word for each example. On average, this group achieved a 36% accuracy rate. While our sample size is indeed limited, and the results may be higher than those of the population at large, we believe this represents a satisfactory benchmark with which to evaluate our model's performance.

Our first experiment attempted to predict the last word of a sentence using a basic LSTM RNN without the attention mechanism. This model constitutes our baseline. To reduce computational effort, we limit RNN unrolling to 40 timesteps for the baseline model and all other extended models presented below. We use the Adam optimization algorithm [13] with a learning rate of 0.01. We also initialize word embeddings with pre-trained 100-dimensional GloVe vectors [14] trained on a Wikipedia corpus. Unless otherwise specified, all models also use a hidden layer size of 128 and a single stacked layer. We incorporate gradient clipping to counter the possibility of exploding gradients by clipping each gradient to a maximum norm of 5.0.

The vanilla RNN performs the task adequately, but does not achieve a cross-entropy validation loss lower than approximately 8.0, which corresponds to an accuracy of 7.02%. See Figure 3 for a graph of the learning progress of the vanilla model.

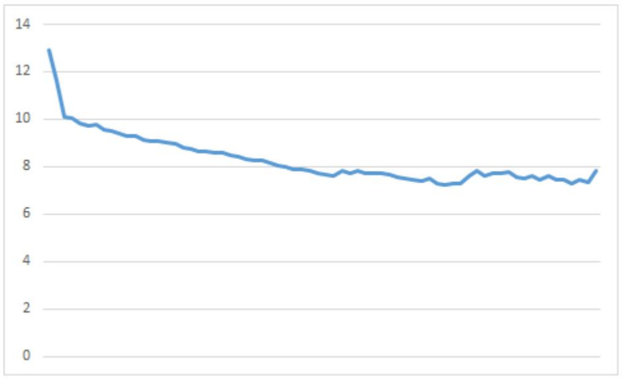

Figure 3: The training progress of the vanilla LSTM RNN. Note that learning plateaus at a validation cross-entropy loss of about 8.0. This corresponds to an accuracy of approximately 7%.

To improve performance, our next experiment incorporated a standard attention mechanism. At first, this addition provided no improvement over the previous vanilla model. Learning still plateaued after a relatively short amount of time. We attempted therefore to incorporate various forms of regularization such as dropout and the addition of an L2 norm penalty. Dropout, however, proved to be remarkably ineffective, and did not mitigate the model's propensity to overfit at all. In fact, increasing the drop probability appears to have increased the level of overfitting, while simultaneously slowing the learning rate.

We met with more success by adding an L2 norm penalty for all trainable variables to the cross-entropy objective function. Although adding a regularization penalty makes the ending loss value somewhat less interpretable, we found that a weight of $1e-5$ resulted in the best accuracy. Accuracy rates for each of the three weights attempted are displayed below.

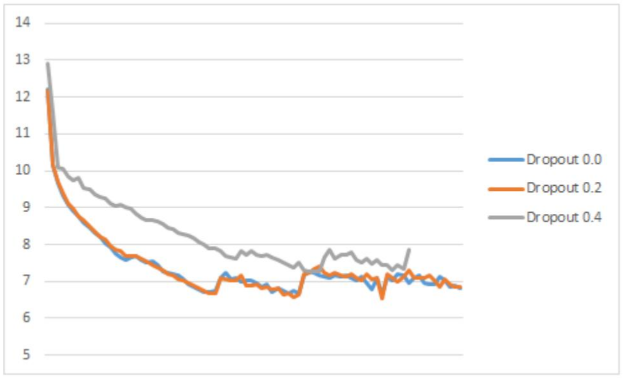

Figure 4: The training progress of an RNN with attention. Note the similarity with the previous figure in that learning plateaus in about the same place.

Table 1: Model accuracy with L2 regularization

WEIGHT	**VALIDATION**
$1e-6$	18.0%
$1e-5$	28.2%
$1e-4$	8.7%

Finally, we present our experiments with cache attention. For all following models, we use the best regularization weight from above, $1e-5$. We experimented with cache sizes of 100, 1,000, and 10,000. The accuracy results are presented below. As our final experiment, we also present accuracy on the test set, which is comparable to the validation accuracy. Only the size 1,000 cache outperforms the standard model with attention, but it does so by a significant margin. We believe that for a cache of size 100, not enough information can be retained to improve performance by a significant degree, while for a cache of size 10,000, overfitting is induced. After all, the size of our training data is only some 80,000 observations, so a cache of size 10,000 stores over 12% of the total data.

Table 2: Model accuracy with caching

CACHE SIZE	**VALIDATION**	**TEST**
100	28.9%	28.1%
1,000	**33.6%**	**32.9%**
10,000	28.7%	28.0%

We also present a visualization of the attention scores for hidden states stored in the cache at the end of training. The results were quite surprising to us. We expected the weights to approximately resemble a step-wise function, where the attention score would gradually increase towards the bottom of the cache. Assuming the min-heap property holds, we should indeed see lower scoring elements towards the top and higher scoring elements towards the bottom. The figure shows that this is partially the case; we observe a subtle step upwards as the cache index increases. However, we also note a more interesting pattern, which is why we chose to present the graph on a logarithmic scale. Observe the regular spacing of the peaks in the distribution, and observe that the distribution is sparse precisely at consecutive powers of two. If we assume that the min-heap property is upheld, this result is unexpected, since we expect elements with the same tree depth to have approximately the same weight. Instead, it is clear that the model assigns a very low weight to elements on the side edges of the tree structure.

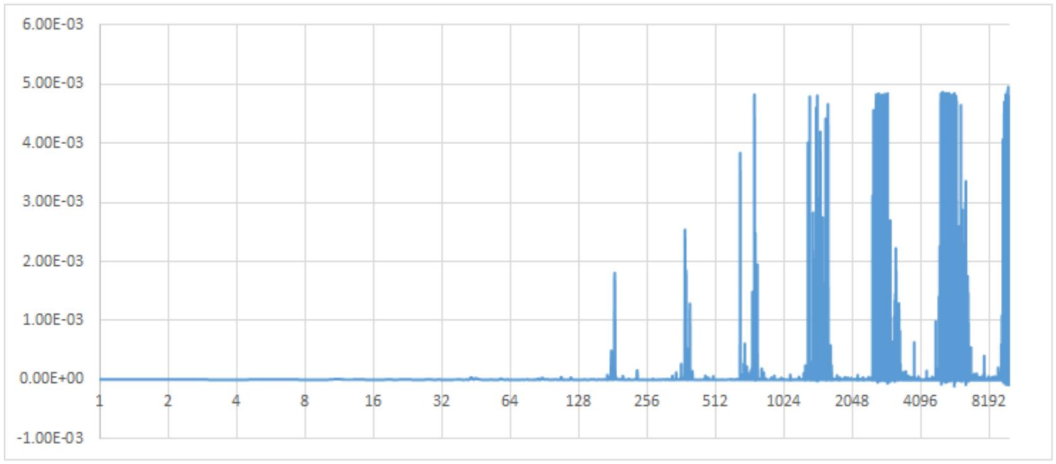

Figure 5: The training progress of an RNN with attention. Note the similarity with the previous figure in that learning plateaus in about the same place.

We lack an adequate explanation for this result, but we propose a probabilistic interpretation. First, note that our implementation is such that at the beginning of training, each successive example is inserted into the heap until there is no more room. Since the cache becomes filled relatively early in the training process, at the point when the cache attention weights are still relatively random, the attention scores of the elements in the cache are essentially random compared to their later values once the weights have been updated. As a result, when inserting a new state into the heap and sifting it down to maintain the min-heap property, it becomes exponentially more likely that the inserted state will land in the center of the tree rather than the edges. As a result, the model quickly learns to a assign a low weight to elements on the sides of the tree.

5 Conclusion and Future Work

In this paper we have shown that an external cached memory has the capacity to improve the performance of deep recurrent language models. Our extension to the construct of a long-term cache uses a standard additive attention mechanism to assign weight to distinct cache entries. This mechanism allows the model to incorporate information from previously seen examples into the current prediction in a more explicit way.

We find that incorporating a cache of size 1,000 (about 1% of the size of our training corpus) improves accuracy on a word prediction task by nearly 5 percentage points relative to a baseline RNN with attention.

In the future, we may seek a more adequate explanation for the intriguing pattern of cache use noted in the section above. Assuming our proposed explanation is correct, a potential solution may be to periodically ”sweep” the cache by removing low weight elements from the sides of the heap. It would also be interesting to apply the same cache attention design to different cache elements. In other words, we could experiment with storing vector representations of n-grams. Our implementation stores sentence representations, but sub-optimally, it may be argued, since the stored representations omit the last word. Instead, perhaps we could store sentence embeddings trained using a ”skip-thought” model.

Please find code at `https://github.com/cpgaffney1/cs224n_proj`.

References

[1] The Unreasonable Effectiveness of RNN’s: http://karpathy.github.io/2015/05/21/ rnn-effectiveness/

[2] Vaswani, Ashish, et al. ”Attention is all you need.” Advances in Neural Information Processing Systems. 2017.

[3] Stanley Chen and Joshua Goodman. 1999. An empirical study of smoothing techniques for language modeling. Computer Speech and Language, 13(4):359393.

[4] G. Zweig, J.C. Platt, C. Meek, C.J.C. Burges, A. Yessenalina, and Q. Liu, Computational approaches to sentence completion, in Proc. Association of Computational Linguistics, 2012.

[5] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. ICLR Workshop, 2013.

[6] Chung, Junyoung, et al. ”Empirical evaluation of gated recurrent neural networks on sequence modeling.” arXiv preprint arXiv:1412.3555 (2014).

[7] Mirowski, Piotr, and Andreas Vlachos. ”Dependency recurrent neural language models for sentence completion.” arXiv preprint arXiv:1507.01193 (2015).

[8] Roland Kuhn. Speech recognition and the frequency of recently used words: A modified markov model for natural language. In Proceedings of the 12th conference on Computational linguistics-Volume 1, 1988.

[9] Grave, E., Joulin, A., and Usunier, N. Improving Neural Language Models with a Continuous Cache. ArXiv e-prints, December 2016b.

[10] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. CoRR, abs/1410.3916, 2014.

[11] Kumar, Ankit, Irsoy, Ozan, Su, Jonathan, Bradbury, James, English, Robert, Pierce, Brian, Ondruska, Peter, Gulrajani, Ishaan, and Socher, Richard. Ask me anything: Dynamic memory networks for natural language processing. 2015.

[12] Zhu, Zhemin, Delphine Bernhard, and Iryna Gurevych. 2010. A monolingual tree-based translation model for sentence simplification. In Proceedings of the 23rd International Conference on Computational Linguistics. Beijing, China, pages 13531361.

[13] Kingma, Diederik P. and Ba, Jimmy. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG], December 2014.

[14] Pennington, Jeffrey, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. Proceedings of the Empiricial Methods in Natural Language Processing (EMNLP 2014) 12.