CACHE BASED RECURRENT NEURAL NETWORK LANGUAGE MODEL INFERENCE FOR FIRST PASS SPEECH RECOGNITION

Zhiheng Huang[⋆] *Geoffrey Zweig*[†] *Benoit Dumoulin*[⋆]

[⋆] Speech at Microsoft, Sunnyvale, CA
[†] Microsoft Research, Redmond, WA
{zhihuang,gzweig,bedumoul}@microsoft.com

ABSTRACT

Recurrent neural network language models (RNNLMs) have recently produced improvements on language processing tasks ranging from machine translation to word tagging and speech recognition. To date, however, the computational expense of RNNLMs has hampered their application to first pass decoding. In this paper, we show that by restricting the RNNLM calls to those words that receive a reasonable score according to a n-gram model, and by deploying a set of caches, we can reduce the cost of using an RNNLM in the first pass to that of using an additional n-gram model. We compare this scheme to lattice rescoring, and find that they produce comparable results for a Bing Voice search task. The best performance results from rescoring a lattice that is itself created with a RNNLM in the first pass.

Index Terms— recurrent neural network language model, cache, computational efficiency, voice search

1. INTRODUCTION

Recurrent neural network language models [1] have been successfully applied in a variety of language processing tasks ranging from machine translation [2] to word tagging [3, 4] and speech recognition [1, 5]. In common with other continuous space language models, RNNLMs map words that are semantically or grammatically related to similar locations in continuous space. Thus, adjusting model parameters for one word in a particular context tends to improve likelihood estimates for similar words in similar contexts. Despite their encouraging performance on a broad array of tasks, RNNLMs have not been widely used for first pass speech recognition because of their computational complexity. Addressing this problem is the topic of this paper.

Previous attempts to use RNN models in speech recognition have fallen in the following two categories. The first class of approaches involves converting an RNN model into an n-gram language model format which can be directly used in first pass decoding. For example, [6] has sampled artificial data from trained RNN models and then trained n-gram models from the sample data. In a similar vein, [7] converted RNN models to weighted finite state transducers, and for feedforward networks, [8] converted NNLM models to n-gram models via a hierarchical pruning algorithm. The second class of approaches involves using RNN models in second pass n-best re-scoring. For example, [9] has proposed to cache redundant computation in n-best hypotheses re-scoring, and [10] has proposed prefix tree based n-best list re-scoring.

In this paper, we address the problem of using RNNLMs directly in the first pass of speech recognition decoding. Our approach has two basic components. First, we use a baseline n-gram model to score search path extensions when words are hypothesized during decoding. Only if a newly hypothesized word has a reasonable score, a RNNLM is called to compute a new LM score, which is then included as part of the path cost. Secondly, we propose a set of caches which store previously computed scores and normalization values. These caches effectively convert the RNN into an n-gram language model dynamically, in a context-dependent way: once a word is hypothesized in a given n-gram context, the probability is reused when the same word is encountered in the same context. In theory, the RNN model is sensitive to the entire path history, but in practice we find that truncating the history and re-using probabilities incurs no penalty.

The main contributions of this paper are: 1) we apply RNN models directly in first pass decoding. For the Bing voice search task, we find that RNN models outperform n-gram models, but at the expense of much greater computational complexity; 2) we propose a cache based RNN inference scheme, which avoids repeated computation of identical LM calls, and caches useful intermediate results for fast RNN inference; and 3) we show that the use of RNN models in both first pass decoding and second pass lattice re-scoring results in the lowest error rate in our task.

The remainder of the paper is organized as follows. In Section 2 we review RNN inference. In Section 3 we describe the decoder which makes calls to RNN inference. We propose cache based RNN inference in Section 4. We report experimental results in Section 5, discuss the potential of our work in Section 6, and draw conclusions in Section 7.

2. RNN INFERENCE

Fig. 1 illustrates an RNN. For simplicity we do not include Maximum Entropy features as proposed in [11]. The network has an input layer $\mathbf{w}(t)$, hidden layer $\mathbf{s}(t)$, output layer class component $\mathbf{c}(t)$, and output layer word component $\mathbf{y}(t)$. The vectors $\mathbf{w}(t)$ and $\mathbf{y}(t)$ have dimensionality of the vocabulary (V). $\mathbf{c}(t)$ and $\mathbf{y}(t)$ represent probability distributions over classes and words respectively given the previous word $\mathbf{w}(t)$

978-1-4799-2893-4/14/$31.00 ©2014 IEEE

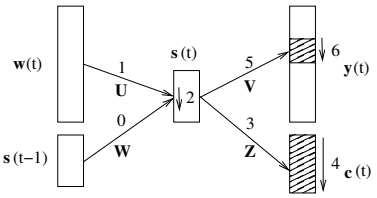

Fig. 1. Recurrent neural network infrastructure. Computational steps are numbered 0 through 6. A downward arrow implies assigning a value to all the nodes in the span of the arrow.

and the hidden state vector $\mathbf{s}(t-1)$.

The values in the hidden and output layers are computed as follows:

$$\mathbf{s}(t) = f(\mathbf{U}\mathbf{w}(t) + \mathbf{W}\mathbf{s}(t-1)), \quad (1)$$

$$\mathbf{c}(t) = g(\mathbf{Z}\mathbf{s}(t)), \quad (2)$$

$$\mathbf{y}(t) = g(\mathbf{V}\mathbf{s}(t)), \quad (3)$$

where $f(z)$ and $g(z)$ are sigmoid and softmax activation functions as following:

$$f(z) = \frac{1}{1+e^{-z}}, \quad (4)$$

$$g(z_m) = \frac{e^{z_m}}{\sum_k e^{z_k}}. \quad (5)$$

The values of $\mathbf{U}$, $\mathbf{V}$, $\mathbf{W}$, and $\mathbf{Z}$ are computed in RNN training. The final probability of output word is the product of $\mathbf{c}(t)$ and $\mathbf{y}(t)$(see [5] for details).

RNN inference consists of 7 steps labeled 0 through 6 in Fig. 1. Steps 0, 1 and 2 correspond to computing $\mathbf{W}\mathbf{s}(t-1)$, $\mathbf{U}\mathbf{w}(t)$, and $f(.)$ respectively in Equation (1). Similarly, steps 3 and 4 correspond to Equation (2), and steps 5 and 6 correspond to Equation (3).

In order to derive the time complexity of RNN inference, we assume an average of $\frac{V}{C}$ words are considered in step 6. This assumption is somewhat more pessimistic than what is obtained in practice with *frequency binning* [5] or *speed-regularized likelihood classing* [12]. Based on this assumption, the average computational complexity of RNN inference, i.e., computing $P(\mathbf{y}(t)|\mathbf{w}(t), \mathbf{s}(t-1)$, is

$$O(H \times H + H \times (C + \frac{V}{C})), \quad (6)$$

where H is the size of hidden layer, V is the size of vocabulary, and C is the size of classes.

3. WEIGHTED FINITE STATE TRANSDUCER (WFST) DECODER

In this section, we briefly describe the decoder which is used in our experiments. It is a dynamic decoder modeled after [13]. The search space is represented as a determinized, minimized finite state transducer. Input arcs are labeled with context dependent HMM states, and output arcs with words. The language model is dynamically applied at word-ends. Language model lookahead scores are embedded in the graph, and compensated for at word ends, when multiple language models may be consulted and interpolated. Based on this organization, we have successfully experimented with the following *skip heuristic* to reduce computation caused by RNN inference: when the cost of a path evaluated by baseline n-gram model is above a threshold, such a path is killed without calling RNN model for evaluation. As we will see in the experiments, the skip heuristic is critical to speed, and does not incur accuracy loss.

In a standard token-passing dynamic WFST decoder using an n-gram language model, tokens landing on the same WFST arc and in the same language model state can be combined. Theoretically, this is incompatible with an RNNLM because each history state depends on the entire path. To resolve this, we introduce a *recombination length* parameter k. Tokens/Paths with identical last k history words are merged. A hidden state of the RNN is stored to represent the k history words (see Section 4.2 for details).

4. CACHE BASED RNN INFERENCE

Four caches are used to speed up RNN inference. The caches are implemented as hash tables to store key value pairs. The caches discussed in this paper are local: they are cleared before decoding a new utterance, and are dynamically constructed as the decoder makes LM calls. The tree on the bottom of Figure 2 shows the topologically ordered RNN LM calls for a given utterance. The numbers in the nodes represent the orders of RNN LM calls. For example, $P(movies| < s >, action)$, is called in the orders of 4 and 6 (duplicated calls are possible during SR decoding). Since the decoder recombines the history, it does make calls that do not involve $< s >$, though these are not shown.

4.1. Cache 1: Query to Language Model Probability Cache

Cache 1 is used to store the LM probability of a LM call. For example, $< s > \ action\ movies \rightarrow P(movies| < s > , action)$ is stored in cache 1 after call #4 and the value is retrieved and re-used for call #6. A hit with cache 1 accomplishes inference with a single hash table lookup.

4.2. Cache 2: History to Hidden State Vector Cache

The history to hidden state vector cache stores key-value pairs of histories and their corresponding hidden state vectors. Algorithm 1 shows cache 2 update as used in previous work such as [9, 10]. An update example of such cache is described as follows. Assume the decoder makes call #4, $P(movies\ |< s >\ action)$, and there is an entry with key being $< s > action$. The value of that key is retrieved to populate $\mathbf{s}(t-1)$ and then steps 0 through 6 are executed in RNN inference. As a result, $P(movies\ |< s >\ action)$ is computed. In addition, we create an entry of $< s >\ action\ movies \rightarrow \mathbf{s}(t)$ if the key of the entry has not been stored in cache2.

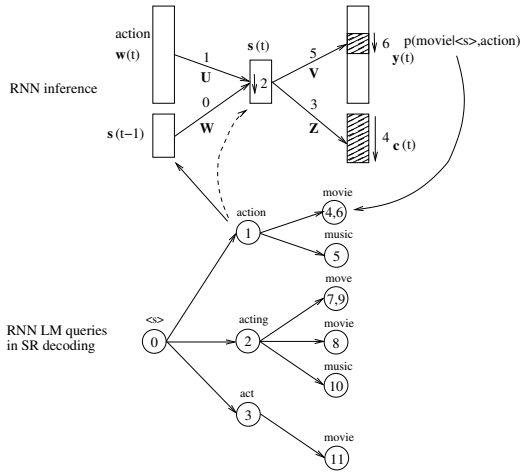

Fig. 2. RNN language model calls in SR decoding

The above cache is denoted as **naive cache 2** as it involves repeated computation when the same history is used to compute the probability of multiple successors: $\mathbf{s}(t-1)$ is repeatedly multiplied by $\mathbf{W}$. To avoid this, in Algorithm 2, we store $\mathbf{Ws}(t-1)$ in Cache 2B. When RNN inference is called, cache2B is first checked. If an entry is found, then steps 0 through 2 can be skipped (complexity savings of $O(H \times H)$ in Equation (6)). Otherwise, the computation is the same as in naive cache 2.

We denote the caching strategies we use as *cache first*. That is, once an entry is stored in cache2, it is not updated. Alternatively, we have tested a *cache best* strategy: we cache the hidden state vector which leads to the highest probability within a sliding time window. Both approaches lead to similar results. We therefore use cache first strategy through out all experiments.

Algorithm 1 Naive cache 2 update algorithm

Ensure: Cache 2 update for query $P(w_i \mid w_{i-n+1} \dots w_{i-1})$
oldHistory = $w_{i-n+1} \dots w_{i-1}$
newHistory = $w_{i-n+2} \dots w_i$
$\mathbf{s}(t-1) = getValue(cache2, oldHistory)$
execute steps 0, 1, and 2 to generate $\mathbf{s}(t)$
if newHistory is not in keys of cache2 **then**
    $cache2(newHistory) = \mathbf{s}(t)$
end if

Algorithm 2 Cache 2 update algorithm

Ensure: Cache2A and cache2B update for query $P(w_i \mid w_{i-n+1} \dots w_{i-1})$
oldHistory = $w_{i-n+1} \dots w_{i-1}$
newHistory = $w_{i-n+2} \dots w_i$
if oldHistory in keys of cache2B **then**
    $\mathbf{s}(t) = getValue(cache2B, oldHistory)$
else
    $\mathbf{s}(t-1) = getValue(cache2A, oldHistory)$
    execute steps 0, 1, and 2 to generate $\mathbf{s}(t)$
    $cache2B(oldHistory) = \mathbf{s}(t)$
end if
if newHistory is not in keys of cache2A **then**
    $cache2A(newHistory) = \mathbf{s}(t)$
end if

4.3. Cache 3: History to Class Normalization Factor Cache

Similar to cache 2, the computation of class probabilities in steps 3 and 4 in RNN inference can be made *once* for calls which have same histories (e.g., $P(movies \mid< s > \ action)$ and $P(music \mid< s > \ action)$). This is because steps 3 and 4 depend on history only ($< s > \ action$ in our examples). We use cache 3 for this purpose. In terms of implementation, we can either cache all classes $P(\mathbf{c}(t) \mid< s > action)$, or cache the class normalization term (the denominator term in Equation (5)). We choose the latter as the former may require significantly more memory. Cache 3 reduces the time complexity factor of $O(H \times C)$ to $O(H)$ in Equation (6) when the key is found in this cache.

4.4. Cache 4: History and Class Id to Sub-vocabulary Normalization Factor Cache

Similar to cache 3, steps 5 and 6 in RNN inference can be computed *once* for calls which have same histories and class id. This is because steps 5 and 6 depend on history and class only. We use cache 4 for this purpose. Again, we store the normalizer only. Cache 4 reduces the complexity factor of $O(H \times \frac{V}{C})$ to $O(H)$ in Equation (6) if the key has been found in this cache.

4.5. Order of Cache Consultation

When queried for a language model probability, we first consult cache 1 for the probability. If it is not found, the probability must be computed. In this case, Caches 2-4 are consulted as inference steps 0-6 are executed.

5. EXPERIMENTS

5.1. Experimental Setting

Table 1. Characteristics of training, validation and test data sets.

data	sentences	tokens
train	549K	2.15M
validation	1344	5316
test	2037	7094

Table 2. Vocabulary size and perplexity on validation and test data sets for baseline, KN4 and RNN models.

	baseline	KN4	RNN
vocab size	70K	35K	35K
validation pplx	156	157	143
test pplx	143	125	120

We apply RNN models to first pass decoding on Bing voice search data [14]. Production acoustic and language

Table 3. First pass word error rates (%) for validation and test data sets using different language models interpolated with the baseline.

	baseline	KN4	RNN	Cache RNN
Valid	26.80	25.90	25.70	25.70
Test	25.30	23.80	23.20	23.20

models were used as the baseline. We used a collection of recently transcribed queries to train our rescoring language models. For RNN model training, we used 50M 3-gram maximum entropy features and 50 classes (see [12], with the speed regularization constant being 0.001). The hidden layer size was 100. Training takes around 3 hours. Table 1 shows the characteristics of the training, validation and test sets.

The baseline n-gram LM model is trained on a much larger data set which consists of heterogeneous data sources such as Bing computer search queries and Bing mobile search queries. Note that the training data do not include the aforementioned in-domain training data. Table 2 shows the vocabulary size and perplexity of the baseline, Kneser-Ney 4-gram (KN4) and RNN language models. The RNN model results in the lowest perplexities for both validation and test data sets.

5.2. First Pass Decoding Results

Table 3 presents word error rates for first pass decoding with our models. The baseline LM has a 25.3% WER on the test set. This falls to 23.2% with the RNNLM. By contrast, using the same data in a KN 4-gram model is 0.6% worse. There is no degradation from caching[1]. All experiments used an interpolation weight of 0.5. Table 4 shows the runtimes; with caching, the use of an RNNLM is comparable to using an additional n-gram LM. This opens avenues in which various syntactic and semantic features [15, 16] can be easily incorporated.

Table 4. First pass speed (xRT) for validation data set using different collections of language models

	baseline	baseline+KN4	baseline+RNN	baseline+Cache RNN
Speed	0.94	1.16	3.49	1.16

Table 5 shows the incremental effect of the four caches as measured on 100 randomly sampled voice search queries. When the RNN is used in every single LM call (i.e. the skip heuristic is not used), and no caching is done, the runtime is over 100xRT. With all our optimizations, this drops to just under 1.2xRT.

Table 5. Incremental speed reduction of four caches. Error rates are the same in all cases.

no skip	skip heuristic					no skip
naive cache 2	naive cache 2	cache 2	+ cache 1	+ cache 3	+ cache 4	all caches
102.77	3.82	3.32	1.32	1.30	1.18	2.75

[1]We use the recombination length of 3, which is enough to preserve word histories as the average voice search query size is under 4.

5.3. Second Pass Lattice Re-scoring Results

For comparison, we have run a set of lattice rescoring experiments. In these experiments, we generate lattices with three language models: baseline, baseline+KN4, and baseline+RNN. In each case we then re-score with the RNN LM. These results are summarized in Table 6. On the test set, rescoring achieves 22.9% compared with 23.2% for first pass decoding. The best results are attained by creating the lattice with the RNNLM interpolated with the baseline ngram model in the first pass, and then rescoring with the same model using interpolation weight 0.3.

Table 6. Word error rates of second pass lattice re-scoring.

	baseline	baseline+KN4	baseline+RNN
RNN weight	0.5	0.4	0.3
Valid WER	25.50	25.50	25.40
Test WER	22.90	23.10	22.70

Table 7. Perplexity on voice search validation and Penn Treebank data sets for cache based RNN models.

	RNN	recombination length				
		1	2	3	4	5
voice search	143.09	190.32	149.96	143.10	143.11	143.09
Penn Treebank	127.29	150.15	150.82	133.45	128.61	127.72

6. DISCUSSION

We have tested our approach with the voice-search task because it is a heavily used and very important speech application. However, it has the property that most utterances are short, and therefore the use of a cache based on a fixed history length might be more effective in this scenario than in others with longer sentences. To test if cache based RNN can be applied to longer utterances, we have computed perplexities for both voice search validation and Penn Treebank sentences. The average voice search utterance is under 4 words, while the average Treebank utterance is 21 words. Table 7 shows perplexities on these two sets, for caching based on history lengths from 1 to 5. We see that in both cases, perplexity drops rapidly up to history lengths 3 or 4, at which point it is very close to that obtained with unlimited context. This suggests that caching may be appropriate for longer utterances.

7. CONCLUSIONS

In this paper, we applied RNN language models directly in first pass decoding. We showed that RNN models outperform n-gram models in the Bing voice search task but with much greater computational complexity. To address this, we propose cache based RNN inference, which can significantly reduce the runtime, while attaining identical results to conventional RNN models. Finally, we showed that the use of RNN models in both first pass decoding and second pass lattice re-scoring results in the lowest WER in our task.

8. REFERENCES

[1] T. Mikolov, M. Karafiat, L. Burget, J. Cernocky, and S. Khudanpur, "Recurrent neural network based language model," in *INTERSPEECH*, 2010.

[2] M. Auli, M. Galley, C. Quirk, and G. Zweig, "Joint language and translation modeling with recurrent neural networks," in *EMNLP*, 2013.

[3] G. Mesnil, X. He, L. Deng, and Y. Bengio, "Investigation of recurrent-neural-network architectures and learning methods for language understanding," in *INTERSPEECH*, 2013.

[4] K. Yao, G. Zweig, M. Hwang, Y. Shi, and Dong Yu, "Recurrent neural networks for language understanding," in *INTERSPEECH*, 2013.

[5] T. Mikolov, S. Kombrink, L. Burget, J. H. Cernocky, and S. Khudanpur, "Extensions of recurrent neural network language model," in *ICASSP*, 2011.

[6] A. Deoras, T. M. Mikolov, S. Kombrink, M. Karafiat, and S. Khudanpur, "Variational approximation of long-span language models for lvcsr," in *ICASSP*, 2011.

[7] G. Lecorve and P. Motlicek, "Conversion of recurrent neural network language models to weighted finite state transducers for automatic speech recognition," in *Interspeech*, 2012.

[8] E. Arisoy, S. F. Chen, B. Ramabhadran, and A. Sethy, "Converting neural network language models into back-off language models for efficient decoding in automatic speech recognition," in *Interspeech*, 2013.

[9] S. Kombrink, T. Mikolov, M. Karafiat, and L. Burget, "Recurrent neural network based language modeling in meeting recognition," in *INTERSPEECH*, 2011.

[10] Y. Si, Q. Zhang, T. Li, J. Pan, and Y. Yan, "Prefix tree based n-best list rescoring for recurrent neural network language model used in speech recognition system," in *Interspeech*, 2013.

[11] T. Mikolov, A. Deoras, D. Povey, L. Burget, and J. H. Cernocky, "Strategies for training large scale neural network languge models," in *ASRU*, 2011.

[12] G. Zweig and K. Makarychev, "Speed regularization and optimality in word classing," in *ICASSP*, 2013.

[13] H. Soltau and G. Saon, "Dynamic network decoding revised," in *ASRU*, 2009.

[14] G. Zweig and S. Chang, "Personalizing model m for voice-search," in *INTERSPEECH*, 2011.

[15] Y. Shi, P. Wiggers, and C. M. Jonker, "Towards recurrent neural networks language models with linguistic and contextual features," in *INTERSPEECH*, 2012.

[16] T. Mikolov and G. Zweig, "Context dependent recurrent neural network language model," in *SLT*, 2012.