JLM - Fast RNN Language Model with Large Vocabulary

Jiali Yao, Marco Fiscato, Katsutoshi Ohtsuki
Microsoft
jiayao@microsoft.com
marco.fiscato@microsoft.com
katsutoshi.ohtsuki@microsoft.com

Xinjian Li
Carnegie Mellon University
xinjianl@andrew.cmu.edu

Abstract

The language model is a key to many tasks like machine translation, speech recognition, and input method. While neural network language model shows better accuracy and scalability to a wider context, the cost to get the probability of next word is non-trivial. Moreover, eastern Asian languages like Japanese and Chinese can easily have a large vocabulary over 100K words to cover the most frequently appeared tokens. In this paper, a fast RNN model named JLM[1] with a hybrid optimization is proposed. The experiment on BCCWJ Japanese corpus shows a 50x speed up during inference with decoder and up to 90% model size reduced without significant perplexity change.

1 Introduction

Statistical language model estimates the probability of a sequence by calculating the probability of the next word giving the context.

$$P(w_0, \ldots, w_1) = \prod_{i=0}^{n} P(w_i | w_o, \ldots, w_{i-1}) \qquad (1)$$

The counted based language model calculates the probability of the next word by counting the frequency of such a context with and without the next word.

$$P(w_i | w_0, \ldots, w_{i-1}) = \frac{count(w_0, \ldots w_{i-1})}{count(w_0, \ldots, w_i)} \qquad (2)$$

It is hard to enumerate all the combinations. In practice, applications will only use tri-gram to avoid exponential growth of storage. Neural network language model with word embeddings, as an alternative, has been studied (Bengio, et al. 2003, Mikolov, et al. 2010). In recent years, on standard benchmarks, RNN language model achieves the state of art perplexity (Jozefowicz, et al. 2016) and outperforms the traditional non-parametric count based language model (Kneser and Ney 1995).

However, comparing to count based n-gram language model, the probability of next word is not a simple table lookup. Instead, the next word probability distribution is computed from context each time. The computation involves a matrix operation of vocabulary size and a final softmax over the vocabulary-sized logits. Techniques like hierarchical softmax (Mnih and Hinton 2008), target sampling (Jean, et al. 2017), and noise contrast estimation (Gutmann and Hyvärinen 2010) can reduce the training time of large vocabulary language model by advanced sampling. But inference time cannot simply be reduced. A fast RNN language model is vital for real-time applications to run in various clients like mobile or PC without GPU acceleration. Also, such applications have a limited budget for model size, sometimes, only a few Mb is allowed for software distribution.

Japanese and Chinese have an additional requirement for high-performance language model as the input is not always segmented words. Conversion or decoding is an essential part. Widely used Viterbi decoder has $O(N \times D^2)$ complexity, where N is number of candidates with same pronunciations, and D is the number of steps. Comparing to an English next word prediction task, a Japanese conversion task is way more expensive.

2 JLM Framework

In this section, an E2E training and decoding framework JLM is proposed. It focuses on inference speed up and optimization on model size.

There are various choices for RNN model. We choose standard LSTM (Hochreiter and Schmidhuber 1997) as the basic benchmark for the improvements. Other architectures include text CNN (Kim 2014), char RNN (Karpathy 2015), and the char aware LSTM (Kim, Jernite, et al. 2015). However, for Japanese language, Kanji char in Unicode is much more than the alphabet in English. Seq2seq model (Sutskever, Vinyals and Le 2014) can also convert Romaji sequence directly to Kanji sequence. But this method lost flexibility on error correction from speech recognition pipeline or user input.

A single layer LSTM with the word as a basic unit is still a practical choice for a high-performance language model as it captures long distance context while not bringing too much architecture complexity to the model.

The word LSTM model can compute the probability distribution over vocabulary V as follows. For a sentence

[1] https://github.com/jiali-ms/JLM

$w_0, w_1, \dots, w_n$ and $w_t \in V$. For each time stamp t, lookup the input embedding $W_{in} \in \mathbb{R}^{d_{in} * |V|}$ for word w_t. The lookup operation produces the word vector x_t for w_t. Standard LSTM cell is then applied to update internal state as equation(3)

$$\begin{aligned} f_t &= \sigma(H_f * h_{t-1} + I_f * x_t + b_f) \\ i_t &= \sigma(H_i * h_{t-1} + I_i * x_t + b_i) \\ o_t &= \sigma(H_o * h_{t-1} + I_o * x_t + b_o) \\ g_t &= tanh(H_g * h_{t-1} + I_g * x_t + b_g) \\ C_t &= f_t \odot C_{t-1} + i_t \odot g_t \\ H_t &= o_t \odot \tanh(C_t) \end{aligned} \tag{3}$$

where f_t, i_t, o_t are forget gate, input gate, and output gate at timestamp t. H, I, b are trainable parameters in the LSTM model.

$$y_t = H_t * W_{out} \tag{4}$$

$$P(w_i|w_0, \dots, w_{i-1}) = softmax(y_t) \tag{5}$$

The hidden state then projects to a $|V|$ size logics with output embedding $W_{out} \in \mathbb{R}^{d_{out} \times |V|}$. And finally, the probability distribution is calculated from softmax with logits y_t. To better describe the performance in experiments, we name the computation in equation (**3**), (**4**), (**5**) as *LSTM cell*, *projection*, *softmax* stages respectively.

2.1 Embedding optimization

For standard LSTM, major parameters are the vocabulary size embeddings $W_{in} \in \mathbb{R}^{d_{in} \times |V|}$ and $W_{out} \in \mathbb{R}^{d_{out} \times |V|}$. Reducing the number of parameters in these two embeddings can shrinks the generated model size and reduces the amount of matrix operations required.

2.1.1 Embedding sharing

Similar words with close input embeddings should have similar probabilities in next word prediction distribution (Press and Wolf 2017). It is a common practice to tie these two embeddings together.

$$W_{in} = W_{out} \tag{6}$$

For the case where d_{in} is 256 and d_{out} is 512, the number of parameters saved is more than a half.

2.1.2 Variable size embeddings

Word with low frequency has less appearance and requires less information to be captured in word embedding (Chen, Grangier and Auli 2015). The words in the vocabulary can be divided into different zones regarding to their frequency. Each of the zones will use only part of the embedding. The algorithm is called differentiated softmax (D-Softmax).

$$e = H * M$$

$$e = (e^0, e^1, \dots, e^n) \tag{7}$$

$$y = (e^0 * W^0, e^1 * W^1, \dots, e^n * W^n)$$

In projection phase, the hidden state is converted to an embedding size vector e by $M \in \mathbb{R}^{d_{out} \times d_{in}}$. The vector e is treated as a concatenation of embeddings of different a zone. Low frequency zones are smaller in size and saves the time for matrix operation. Note that $W^i \in \mathbb{R}^{d_{e^i} \times |V^i|}$, $|V| = \sum |V^i|$, and $d_{out} = \sum |d_{e^i}|$.

A variation to the differentiated softmax, named D-softmax* (Grave, et al. 2017) keeps the original embedding by projecting each zone with a small matrix as equation shows.

$$y = (e * P^0 * W^0, \dots, e * P^n * W^n) \tag{8}$$

where $e \in \mathbb{R}^{1 \times d_{out}}$, $P^i \in \mathbb{R}^{d_{out} \times d_{e^i}}$. We compared the performance of both methods in the experiment.

2.1.3 Embedding compression

The trained weights can be further compressed to save model size or even runtime memory footprint. Methods like network pruning (Wen, et al. 2016) and quantization (Chen, et al. 2015) greatly reduce the size of the model without loss accuracy for certain tasks. (Shu and Nakayama 2017) proposed a method that can compress model more than 90% using codebook. It assumes similar words do not require the tiny differences in the long vector to capture. Any embedding can be approximated with M embeddings in a codebook C.

$$x(C_w) = \sum_{i=1}^{M} x(C_w^i) \tag{9}$$

Applying this method to language model requires a pre-trained model to start with. After the embeddings are compressed with the codebook, use the original model to refine the rest of the parameters.

2.2 Decoder optimization

The decoder is a vital part of speech recognition and input where observations are decoded from the hidden states. In language model, the search space is dominated by the different words with the same reading.

The top 5 readings for Japanese is listed in the following table. The search space is too big to be effective for a decoder.

Table 1. Top popular readings in BCCWJ corpus.

テン	マン	イチ	ニ	ゴ
10228	7173	7161	6117	5813

2.2.1 *Beam search*

It is common to assume the possible path are from the best subpaths. Beam search help regularize the search space to $O(B \times D \times N)$ where B is the beam size. The inference required is fixed $O(B \times D)$ times.

2.2.2 *Batch decoding*

An important trick in decoding is batching. The underline matrix library takes a batch of prediction task just as a matrix with one more dimension. Instead of doing next word prediction for each path, concatenate all the prediction in the current subpaths will accelerate the speed several times.

3 Experiment

In this section, we analyzed performance of all the optimizations implemented the JLM in detail.

3.1 BCCWJ Corpus

BCCWJ [2] (Balanced Corpus of Contemporary Written Japanese) is a corpus with various source and represent the current written Japanese. The corpus disk vol2 contains a segmented format in short unit words. We parsed the format and build a lexicon that has each word as Display/Reading/POS, e.g. “言語/ゲンゴ/名詞-普通名詞-一般”. The corpus has 127M tokens and 5.8M sentences. It contains 611K words in above definition. The token coverage with top frequent words is analyzed as a reference to mark unknown words. We choose 50K and 100K top frequently used words as the vocab with word embeddings in the experiments.

Table 2. Token coverage with selected words

Selected vocab size	Token coverage
10K	91.0%
50K	97.3%
100K	98.7%

3.2 Experiment setup

The training is done with TensorFlow in Nvidia 1080 GPU card. The trained weights are then dumped. The inference performance is measure with numpy on a machine with Intel E5 CPU. The matrix operation acceleration relies on the underline BLAS library. We believe it is comparable to other solution like Eigen with C++.

For language model evaluation, the standard LSTM with a medium size embeddings 256 and hidden size 512 is chosen as the baseline (LSTM-base). we compared the overall performance among the LSTM-base, the shared embedding version (Tie-embedding), the differential softmax (D-softmax), and the variation of differential softmax (D-softmax*). The D-softmax has the segmentation to corpus as 12%, 18%, 70%. And the embedding sizes are 200, 100, 50 each section accordingly. Since the D-softmax has much fewer parameters than baseline, we also created a LSTM in small size (LSTM-S) with the same number of parameters to D-softmax.

3.3 Language model evaluation

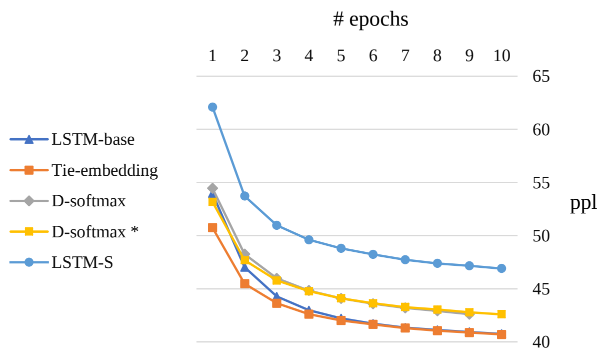

Figure 1. The validation perplexity drops as training epochs. The x-axis is the epochs. Numbers in y-axis are perplexity. The experiment is done with a 50K vocabulary size.

The experiments show that advanced optimization like tie-embedding, D-softmax, and its variation can reach the similar perplexity with LSTM baseline. LSTM-S instead cannot converge to a low perplexity as others. It proves that simply reducing the hyperparameters cannot lead to an optimized language model with good perplexity.

We then tested the inference speed for each model by taking the average of 100 times next word prediction. The LSTM cell is the recurrent part that computes the next hidden state. It is stable for various kinds of vocabulary size. Its speed is dominated by the hidden and embedding sizes. The projection instead is the most time-consuming stage and sensitive to size of vocabulary.

Table 3. Average inference time (ms) for 50K vocab.

	LSTM-base	Tie-Embedding	D-softmax	D-softmax*
LSTM cell	0.535	0.523	**0.755**	0.426
projection	9.832	5.168	1.531	1.461
Softmax	0.8	0.9	0.73	0.66
Total	11.167	6.591	3.016	**2.547**

Table 4. Average inference time (ms) for 100K vocab.

	LSTM-base	Tie-Embedding	D-softmax	D-softmax*
LSTM cell	0.507	0.544	**0.663**	0.419
projection	18.744	9.708	**2.886**	3.002
softmax	0.214	0.16	0.15	0.151
total	19.465	10.412	3.699	**3.572**

[2] http://pj.ninjal.ac.jp/corpus_center/bccwj/en/

The results in the above tables show that increasing the vocabulary size will linearly increase the inference time cost. With D-softmax, the inference time is about **5-6 times** faster than baseline. We notice that the D-softmax* has a smaller embedding size and outperforms the D-softmax.

Table 5. Overall comparison of quality and performance.

	perplexity		model size (mb)		Inference (ms)	
	50k	100k	50k	100k	50k	100k
LSTM-base	40.8	46.0	156.5	306.7	11.7	19.5
Tie-emb	40.7	45.9	56.9	107.1	6.6	10.4
D-softmax	42.6	47.2	22.9	38.1	3.0	3.7
D-softmax*	42.6	47.3	**21.5**	**36.7**	2.5	3.6
Comp-emb	54.0		**10.49**			

Finally, the perplexity, model size, and inference speed are listed together in table 5. The model size can be reduced by **86%** and **88%** for 50K and 100K with D-softmax*. It also has a correlation with the inference speed up. The embedding compressed experiment (Comp-emb) based on Tie-embedding result finally gain a 93% model size reduction. However, its perplexity it not yet optimized. It is useful for the cases where perplexity can be sacrificed.

3.4 Evaluation with decoder

The overall inference speed with the decoder is compared between the LSTM baseline and the D-softmax*. We use an example sentence "きょうはいいてんき". It has 10 frames including a sentence start.

Table 6. Decoding time comparison in seconds.

	LSTM-base		D-softmax*	
	Batch	No-batch	Batch	No-batch
Beam 1	1.2 (10)	1.107(10)	0.19(10)	0.2 (10)
Beam 10	2.1 (10)	9.5 (86)	0.41 (10)	1.7 (86)
Beam 50	2.4 (10)	**47.1 (397)**	**0.92 (10)**	7.8 (397)

The number in the parentheses are the times that inference is called. Since batch will make one call each frame, the number is constantly 10. An optimized batch D-softmax* is **50x** faster than no-batch LSTM baseline with beam width 50.

4 Conclusion

The work shows for large vocabulary language model, standard RNN model like LSTM is not sufficient for product requirements. Optimizations to reduce the final projection phase is the key to reduce the cost. Further optimization on CPU cache and multi-core parallelism may accelerate inference speed more.

References

Bengio, Yoshua, R´ejean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. "A Neural Probabilistic Language Model." *Journal of Machine Learning Research* 3:1137–1155.

Chen, Welin, David Grangier, and Michael Auli. 2015. *Strategies for training large vocabulary neural language models.* arXiv preprint arXiv:1512.04906.

Chen, Wenlin, James T. Wilson, Stephen Tyree, Kilian Q. Weinberger, and Chen Yixin. 2015. "Compressing neural networks with the hashing trick." *ICML.*

Grave, Edouard, Armand Joulin, Moustapha Cissé, David Grangier, and Hervé Jégou. 2017. "Efficient softmax approximation for GPUs." *ICML.*

Gutmann, Michael, and Aapo Hyvärinen. 2010. "Noise-contrastive estimation: A new estimation principle for unnormalized statistical models." *PMLR.* 9:297-304.

Hochreiter, Sepp, and Jürgen Schmidhuber. 1997. "Long short-term memory." *Neural computation* 1735--1780.

Jean, Sébastien, Kyunghyun Cho, Roland Memisevic, and Yoshua Bengio. 2017. "On Using Very Large Target Vocabulary for Neural Machine Translation." *CoRR.*

Jozefowicz, Rafal, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. 2016. "Exploring the Limits of Language Modeling." *arVix eprint arXiv:1602.02410.*

Karpathy, Andrej. 2015. *http://karpathy.github.io/2015/05/21/rnn-effectiveness/.*

Kim, Yoon. 2014. "Convolutional Neural Networks for Sentence Classification." *EMNLP.*

Kim, Yoon, Yacine Jernite, David Sontag, and Alexander M. Rush. 2015. "Character-Aware Neural Language Models." *AAAI.*

Kneser, Reinhard, and Hermann Ney. 1995. "Improved backing-off for m-gram language modeling." *ICASSP.*

Mikolov, Tomas, Martin Karafiát, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. 2010. "Recurrent neural network based language model." *INTERSPEECH.* 1045-1048.

Mnih, Andriy, and Geoffrey E. Hinton. 2008. "A Scalable Hierarchical Distributed Language Model." *NIPS.*

Press, Ofir, and Lior Wolf. 2017. "Using the Output Embedding to Improve Language Models." *EACL.*

Shu, Raphael, and Hideki Nakayama . 2017. *Compressing Word Embeddings via Deep Compositional Code Learning.* arXiv preprint arXiv:1711.01068.

Sutskever, Ilya, Oriol Vinyals, and Quoc V. Le. 2014. "Sequence to Sequence Learning with Neural Networks." *arVix eprint arXiv:1409.3215.*

Wen, Wei, Chunpeng Wu, Yandan Wang, Yiran Chen, and Li Hai. 2016. "Learning structured sparsity in deep neural networks." *NIPS.*