EXTENSIONS OF RECURRENT NEURAL NETWORK LANGUAGE MODEL

Tomáš Mikolov[1,2], *Stefan Kombrink*[1], *Lukáš Burget*[1], *Jan "Honza" Černocký*[1], *Sanjeev Khudanpur*[2]

[1]Brno University of Technology, Speech@FIT, Czech Republic
[2] Department of Electrical and Computer Engineering, Johns Hopkins University, USA
{imikolov,kombrink,burget,cernocky}@fit.vutbr.cz, khudanpur@jhu.edu

ABSTRACT

We present several modifications of the original recurrent neural network language model (RNN LM). While this model has been shown to significantly outperform many competitive language modeling techniques in terms of accuracy, the remaining problem is the computational complexity. In this work, we show approaches that lead to more than 15 times speedup for both training and testing phases. Next, we show importance of using a backpropagation through time algorithm. An empirical comparison with feedforward networks is also provided. In the end, we discuss possibilities how to reduce the amount of parameters in the model. The resulting RNN model can thus be smaller, faster both during training and testing, and more accurate than the basic one.

Index Terms— language modeling, recurrent neural networks, speech recognition

1. INTRODUCTION

Statistical models of natural language are a key part of many systems today. The most widely known applications are automatic speech recognition (ASR), machine translation (MT) and optical character recognition (OCR). In the past, there was always a struggle between those who follow the statistical way, and those who claim that we need to adopt linguistics and expert knowledge to build models of natural language. The most serious criticism of statistical approaches is that there is no true understanding occurring in these models, which are typically limited by the Markov assumption and are represented by n-gram models. Prediction of the next word is often conditioned just on two preceding words, which is clearly insufficient to capture semantics. On the other hand, the criticism of linguistic approaches was even more straightforward: despite all the efforts of linguists, statistical approaches were dominating when performance in real world applications was a measure.

Thus, there has been a lot of research effort in the field of statistical language modeling. Among models of natural language, neural network based models seemed to outperform most of the competition [1] [2], and were also showing steady improvements in state of the art speech recognition systems [3]. The main power of neural network based language models seems to be in their simplicity: almost the same model can be used for prediction of many types of signals, not just language. These models perform implicitly **clustering of words in low-dimensional space**. Prediction based on this compact representation of words is then more robust. No additional smoothing of probabilities is required.

This work was partly supported by European project DIRAC (FP6-027787), Grant Agency of Czech Republic project No. 102/08/0707, Czech Ministry of Education project No. MSM0021630528 and by BUT FIT grant No. FIT-10-S-2.

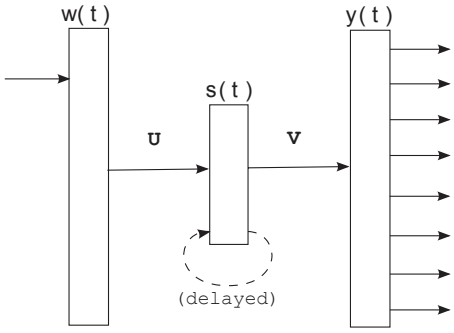

Fig. 1. *Simple recurrent neural network.*

Among many following modifications of the original model, the recurrent neural network based language model [4] provides further generalization: instead of considering just several preceding words, neurons with input from recurrent connections are assumed to represent **short term memory**. The model learns itself from the data how to represent memory. While shallow feedforward neural networks (those with just one hidden layer) can only cluster similar words, recurrent neural network (which can be considered as a deep architecture [5]) can perform **clustering of similar histories**. This allows for instance efficient representation of patterns with variable length.

In this work, we show the importance of the Backpropagation through time algorithm for learning appropriate short term memory. Then we show how to further improve the original RNN LM by decreasing its computational complexity. In the end, we briefly discuss possibilities of reducing the size of the resulting model.

2. MODEL DESCRIPTION

The recurrent neural network described in [4] is also called Elman network [6]. Its architecture is shown in Figure 1. The vector $\mathbf{x}(t)$ is formed by concatenating the vector $\mathbf{w}(t)$ that represents the current word while using 1 of N coding (thus its size is equal to the size of the vocabulary) and vector $\mathbf{s}(t-1)$ that represents output values in the hidden layer from the previous time step. The network is trained by using the standard backpropagation and contains input, hidden and output layers. Values in these layers are computed as follows:

$$\mathbf{x}(t) = [\mathbf{w}(t)^T \mathbf{s}(t-1)^T]^T \tag{1}$$

$$s_j(t) = f\left(\sum_i x_i(t) u_{ji}\right) \tag{2}$$

$$y_k(t) = g\left(\sum_j s_j(t) v_{kj}\right) \tag{3}$$

Table 1. *Comparison of different language modeling techniques on Penn Corpus. Models are interpolated with KN backoff model.*

Model	PPL
KN5	141
Random forest (Peng Xu) [8]	132
Structured LM (Filimonov) [9]	125
Syntactic NN LM (Emami) [10]	107
RNN trained by BP	113
RNN trained by BPTT	106
4x RNN trained by BPTT (mixture)	98

where $f(z)$ and $g(z)$ are sigmoid and softmax activation functions (the softmax function in the output layer is used to make sure that the outputs form a valid probability distribution, i.e. all outputs are greater than 0 and their sum is 1):

$$f(z) = \frac{1}{1+e^{-z}}, \quad g(z_m) = \frac{e^{z_m}}{\sum_k e^{z_k}} \tag{4}$$

The cross entropy criterion is used to obtain an error vector in the output layer, which is then backpropagated to the hidden layer. The training algorithm uses validation data for early stopping and to control learning rate. Training iterates over all the training data in several epochs before convergence is achieved - usually, 10-20 epochs are needed. However, a valid question is whether the simple backpropagation (BP) is sufficient to train the network properly - if we assume that the prediction of the next word is influenced by information which was present several time steps back, there is no guarantee that the network will learn to keep this information in the hidden layer. While the network can remember such information, it is more by luck than by design.

3. BACKPROPAGATION THROUGH TIME

Backpropagation through time (BPTT) [11] can be seen as an extension of the backpropagation algorithm for recurrent networks. With truncated BPTT, the error is propagated through recurrent connections back in time for a specific number of time steps (here referred to as τ). Thus, the network learns to remember information for several time steps in the hidden layer when it is learned by the BPTT. Additional information and practical advices for implementation of BPTT algorithm are described in [7].

The data used in the following experiments were obtained from Penn Tree Bank: sections 0-20 were used as training data (about 930K tokens), sections 21-22 as validation data (74K) and sections 23-24 as test data (82K). The vocabulary is limited to 10K words. The processing of the data is exactly the same as used by [10] and other researchers. For a comparison of techniques, see Table 1. KN5 denotes the baseline: interpolated 5-gram model with modified Kneser Ney smoothing and no count cutoffs.

To improve results, it is often better to train several networks (that differ either in random initialization of weights or also in the numbers of parameters) than having one huge network. The combination of these networks is done by linear interpolation with equal weights assigned to each model (note similarity to random forests that are composed of different decision trees [8]). The combination of various amounts of models is shown in Figure 2.

Figure 3 shows the importance of number of time steps τ in BPTT. To reduce noise, results are reported as an average of perplexity given by four models with different RNN configurations (250, 300, 350 and 400 neurons in the hidden layer). Also, a combination of these models is shown (again, linear interpolation was used). As can be seen, 4-5 steps of BPTT training seems to be sufficient. Note that while complexity of the training phase increases with the amount of steps for which the error is propagated back in time, the complexity of the test phase is constant.

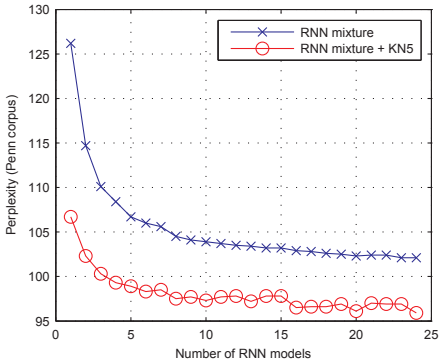

Fig. 2. *Linear interpolation of different RNN models trained by BPTT.*

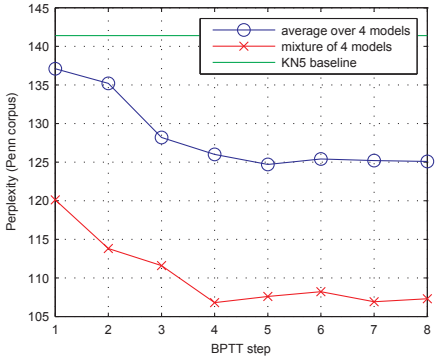

Fig. 3. *Effect of BPTT training on Penn Corpus. BPTT=1 corresponds to standard backpropagation.*

Table 2 shows comparison of the feedforward [12], simple recurrent [4] and BPTT-trained recurrent neural network language models on two corpora. Perplexity is shown on the test sets for configurations of networks that were working the best on the development sets. We can see that the simple recurrent neural network already outperforms the standard feedforward network, while BPTT training provides another significant improvement.

Table 2. *Comparison of different neural network architectures on Penn Corpus (1M words) and Switchboard (4M words).*

	Penn Corpus		Switchboard	
Model	NN	NN+KN	NN	NN+KN
KN5 (baseline)	-	141	-	92.9
feedforward NN	141	118	85.1	77.5
RNN trained by BP	137	113	81.3	75.4
RNN trained by BPTT	123	106	77.5	72.5

4. SPEEDUP TECHNIQUES

The time complexity of one training step is proportional to

$$O = (1 + H) \times H \times \tau + H \times V \quad (5)$$

where H is the size of the hidden layer, V size of the vocabulary and τ the amount of steps we backpropagate the error back in time[1]. Usually $H << V$, so the computational bottleneck is between the hidden and output layers. This has motivated several researchers to investigate possibilities how to reduce this huge weight matrix. Originally, Bengio [1] has merged all low frequency words into one special token in the output vocabulary, which usually results in 2-3 times speedup without significant degradation of the performance. This idea was later extended - instead of using unigram distribution for words that belong to the special token, Schwenk [3] used probabilities from a backoff model for the rare words.

An even more promising approach was based on the assumption that words can be mapped to classes [13] [14]. If we assume that each word belongs to exactly one class, we can first estimate the probability distribution over the classes using RNN and then compute the probability of a particular word from the desired class while assuming unigram distribution of words within the class:

$$P(w_i|history) = P(c_i|history)P(w_i|c_i) \quad (6)$$

This reduces computational complexity to

$$O = (1 + H) \times H \times \tau + H \times C, \quad (7)$$

where C is the number of classes. While this architecture has obvious advantages over the previously mentioned approaches as C can be order of magnitude smaller than V without sacrificing much of accuracy, the performance depends heavily on our ability to estimate classes precisely. The classical Brown clustering is usually not very useful, as its computational complexity is too high and it is often faster to estimate the full neural network model.

4.1. Factorization of the output layer

We can go further and assume that the probabilities of words within a certain class do not depend just on the probability of the class itself, but also on the history - in context of neural networks, that is the hidden layer $\mathbf{s}(t)$. We can change Equation 6 to

$$P(w_i|history) = P(c_i|\mathbf{s}(t))P(w_i|c_i, \mathbf{s}(t)) \quad (8)$$

The corresponding RNN architecture is shown in Figure 4. This idea has been already explored by Morin [13] (and in the context of Maximum Entropy models by Goodman [14]), who extended it further by assuming that the vocabulary can be represented by a hierarchical binary tree. The drawback of Morin's approach was the dependence on WordNet for obtaining word similarity information, which can be unavailable for certain domains or languages.

In our work, we have implemented simple factorization of the output layer using classes. Words are assigned to classes proportionally, while respecting their frequencies (this is sometimes referred to as 'frequency binning'). The amount of classes is a parameter. For example, if we choose 20 classes, words that correspond to the first 5% of the unigram probability distribution would be mapped to class 1 (with Penn Corpus, this would correspond to token 'the' as its unigram probability is about 5%), the words that correspond to the next 5% of the unigram probability mass would be mapped to class 2, etc. Thus, the first classes can hold just single words, while the last classes cover thousands of low-frequency words[2].

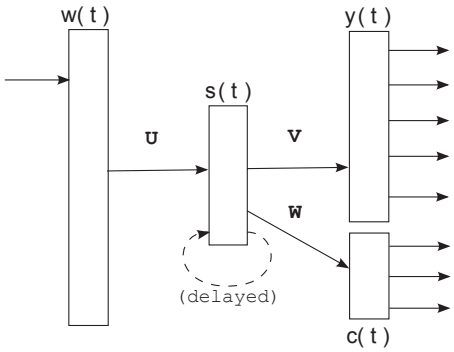

Fig. 4. *RNN with output layer factorized by class layer.*

Instead of computing a probability distribution over all words as it is specified in (3), we first estimate a probability distribution over the classes and then a distribution over the words from a single class, the one that contains the predicted word:

$$c_l(t) = g\left(\sum_j s_j(t) w_{lj}\right) \quad (9)$$

$$y_c(t) = g\left(\sum_j s_j(t) v_{cj}\right) \quad (10)$$

The activation function g for both these distributions is again softmax (Equation 4). Thus, we have the probability distribution both for classes and for words within class that we are interested in, and we can evaluate Equation 8. The error vector is computed for both distributions and then we follow the backpropagation algorithm, so the errors computed in the word-based and the class-based parts of the network are summed together in the hidden layer. The advantage of this approach is that the network still uses the whole hidden layer to estimate a (potentially) full probability distribution over the full vocabulary, while factorization allows us to evaluate just a subset of the output layer both during the training and during the test phases. Based on the results shown in Table 3, we can conclude that fast evaluation of the output layer via classes leads to around 15 times speedup against model that uses full vocabulary (10K), at a small cost of accuracy. The non-linear behaviour of reported time complexity is caused by the constant term $(1+H) \times H \times \tau$ and also by suboptimal usage of cache with large matrices. With $C = 1$ and $C = V$, the model is equivalent to the full RNN model.

4.2. Compression layer

Alternatively, we can think about the two parts of the original recurrent network separately: first, there is a matrix **U** responsible for the input and for the recurrent connections that maintain short term

[1] As suggested to us by Y. Bengio, the τ term can practically disappear from the computational complexity, provided that the update of weights is not done at every time step [11].

[2] After this paper was written, we have found that Emami [18] has proposed a similar technique for reducing computational complexity, by assigning words into statistically derived classes. The novelty of our approach is thus in showing that simple frequency binning is adequate to obtain reasonable performance.

Table 3. *Perplexities on Penn corpus with factorization of the output layer by the class model. All models have the same basic configuration (200 hidden units and BPTT=5). The Full model is a baseline and does not use classes, but the whole 10K vocabulary.*

Classes	RNN	RNN+KN5	Min/epoch	Sec/test
30	134	112	12.8	8.8
50	136	114	9.8	6.7
100	136	114	9.1	5.6
200	136	113	9.5	6.0
400	134	112	10.9	8.1
1000	131	111	16.1	15.7
2000	128	109	25.3	28.7
4000	127	108	44.4	57.8
6000	127	109	70	96.5
8000	124	107	107	148
Full	123	106	154	212

memory, and then a matrix **V** that is used to obtain probability distribution in the output layer. Both weight matrices share the same hidden layer, however, while matrix **U** needs this vector to maintain all short term memory to store information for possibly several time steps, matrix **V** needs only the information contained in the hidden layer that is needed to calculate probability distribution for the immediately following word[3]. To reduce the size of the weight matrix **V**, we can use an additional **compression layer** between the hidden and output layers. We have used sigmoid activation function for the compression layer, thus this projection is non-linear.

A compression layer not only reduces computational complexity, but also reduces the total amount of parameters, which results in more compact models. It is also possible to use a similar compression layer between input and hidden layer to further reduce the size of the models (such layer is usually referred to as a **projection layer**). The empirical results show that with growing amount of training data, the hidden layer needs to be increased to allow the model to store more information. Thus, the idea of using a compression layer is mostly useful when large amount of training data is used. We plan to report results with compression layers in the future.

5. CONCLUSION AND FUTURE WORK

We presented to our knowledge the first published results when using RNN trained by BPTT in the context of statistical language modeling. The comparison to standard feedforward neural network based language models, as well as comparison to BP trained RNN models shows clearly the potential of the presented model. Furthermore, we have shown how to obtain significantly better accuracy of RNN models by combining them linearly. The resulting mixture of RNN models attains perplexity 96 on the well-known Penn corpus, which is significantly better than the best previously published result on this setup [10]. In the future work, we plan to show how to further improve accuracy by combining statically and dynamically evaluated RNN models [4] and by using complementary language modeling techniques to obtain even much lower perplexity. In our ongoing ASR experiments, we have observed good correlation between perplexity improvements and word error rate reduction.

Next, we have shown several possibilities how to reduce computational and space complexity by using classes, factorization of the output layer and by using compression layers. Combinations of these techniques lead to efficient training on very large corpora - we plan to describe our current experiments that involve models trained on much more than 100M words while using non-truncated vocabulary.

Finally, we plan to show that the resulting models can be efficiently used in state of the art systems that use very good baseline acoustic and language models based on huge amounts of in-domain data, and that the additional processing cost by using RNN models does not need to be impractically high by exploiting techniques described in this paper. For that purpose, we published a freely available toolkit for training RNN language models which is available at `http://www.fit.vutbr.cz/~imikolov/rnnlm/`.

[3] Alternatively, we can ask if the rank of the matrix **V** is full.

6. REFERENCES

[1] Yoshua Bengio, Rejean Ducharme and Pascal Vincent. 2003. A neural probabilistic language model. Journal of Machine Learning Research, 3:1137-1155

[2] Joshua T. Goodman (2001). A bit of progress in language modeling, extended version. Technical report MSR-TR-2001-72.

[3] Holger Schwenk, Jean-Luc Gauvain. Training Neural Network Language Models On Very Large Corpora. in Proc. Joint Conference HLT/EMNLP, October 2005.

[4] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černocký, Sanjeev Khudanpur: Recurrent neural network based language model, In: Proc. INTERSPEECH 2010

[5] Y. Bengio, Y. LeCun. Scaling learning algorithms towards AI. In Large-Scale Kernel Machines, MIT Press, 2007.

[6] Jeffrey L. Elman. Finding Structure in Time. 1990. Cognitive Science, 14, 179-211

[7] Mikael Bodén. A Guide to Recurrent Neural Networks and Backpropagation. In the Dallas project, 2002.

[8] Peng Xu. Random forests and the data sparseness problem in language modeling, Ph.D. thesis, Johns Hopkins University, 2005.

[9] Denis Filimonov and Mary Harper. 2009. A joint language model with fine-grain syntactic tags. In EMNLP.

[10] Ahmad Emami, Frederick Jelinek. Exact training of a neural syntactic language model. In ICASSP 2004.

[11] D. E. Rumelhart, G. E. Hinton, R. J. Williams. 1986. Learning internal representations by back-propagating errors. Nature, 323:533.536.

[12] Tomáš Mikolov, Jiří Kopecký, Lukáš Burget, Ondřej Glembek and Jan Černocký: Neural network based language models for highly inflective languages, In: Proc. ICASSP 2009.

[13] F. Morin, Y. Bengio: Hierarchical Probabilistic Neural Network Language Model. AISTATS'2005.

[14] J. Goodman. Classes for fast maximum entropy training. In: Proc. ICASSP 2001.

[15] A. Alexandrescu, K. Kirchhoff. 2006. Factored neural language models. In HLT-NAACL.

[16] Yoshua Bengio and Patrice Simard and Paolo Frasconi. Learning Long-Term Dependencies with Gradient Descent is Difficult. IEEE Transactions on Neural Networks, 5, 157-166.

[17] Y. Bengio, J.-S. Senecal. Adaptive Importance Sampling to Accelerate Training of a Neural Probabilistic Language Model. IEEE Transactions on Neural Networks, 2008.

[18] Ahmad Emami. A Neural Syntactic Language Model. Ph.D. thesis, Johns Hopkins University, 2006.