Wright State University
CORE Scholar

Browse all Theses and Dissertations | Theses and Dissertations

2018

Slim Embedding Layers for Recurrent Neural Language Models

Zhongliang Li
Wright State University

Follow this and additional works at: https://corescholar.libraries.wright.edu/etd_all

 Part of the Computer Engineering Commons, and the Computer Sciences Commons

Repository Citation
Li, Zhongliang, "Slim Embedding Layers for Recurrent Neural Language Models" (2018). *Browse all Theses and Dissertations*. 1992.
https://corescholar.libraries.wright.edu/etd_all/1992

This Dissertation is brought to you for free and open access by the Theses and Dissertations at CORE Scholar. It has been accepted for inclusion in Browse all Theses and Dissertations by an authorized administrator of CORE Scholar. For more information, please contact corescholar@www.libraries.wright.edu, library-corescholar@wright.edu.

Slim Embedding Layers for Recurrent Neural Language Models

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

by

Zhongliang Li
B.A., Peking University, 2011

2018
Wright State University

Wright State University
Graduate School

July 24, 2018

I HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER MY SUPERVISION BY Zhongliang Li ENTITLED Slim Embedding Layers for Recurrent Neural Language Models BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Doctor of Philosophy.

Michael Raymer, Ph.D.
Advisor

Shaojun Wang, Ph.D.
Co-advisor

Michael Raymer, Ph.D.
Director, Computer Science and Engineering
Ph.D. Program

Barry Milligan, Ph.D.
Interim Dean of the Graduate School

Committee on
Final Examination

Jack Jean, Ph.D.

Xinhui Zhang, Ph.D.

Krishnaprasad Thirunarayan, Ph.D.

ABSTRACT

Li, Zhongliang. Ph.D., Department of Computer Science and Engineering, Wright State University, 2018. *Slim Embedding Layers for Recurrent Neural Language Models.*

Recurrent neural language (RNN) models are the state-of-the-art method for language modeling. When the vocabulary size is large, the space taken to store the model parameters becomes the bottleneck for the use of these type of models. We introduce a simple space compression method that stochastically shares the structured parameters at both the input and output embedding layers of RNN models to significantly reduce the size of model parameters, but still compactly represents the original input and the output embedding layers. The method is easy to implement and tune. Experiments on several data sets show that the new method achieves perplexity and BLEU score results comparable to the best existing methods, while only using a tiny fraction of the parameters required by other approaches.

Contents

List of Figures

List of Tables

Acknowledgment

First, I would like to extend my thanks to my advisor Dr. Shaojun Wang. Through our discussion, I realized the essential nature of critical thinking is necessary to have independent opinions of our own and not to merely follow the herd.

Second, I want to thank Dr. Michael Raymer. During the last few years of my study while on Dr. Raymer's team, his support helped me find the confidence to finish my research. I want to thank Dr. Yunxin Zhao for guiding me to investigate the clustering based assignment of sub-vectors. Finally I would like to give my gratitude to Dr. Jack Jean, Dr. Krishnaprasad Thirunarayan, and Dr. Xinhui Zhang for their advice and for serving on my committee.

During my graduate study, I have made many friends. I want to thank my coworkers from the same lab, Tian Xia, Shaodan Zhai, Raymond Kulhanek, Ming Tan, etc. During this graduate study, they were helpful with my study and research, and also with advice for my life.

Dedicated to my parents

Introduction to Statistical Language Modeling

Statistical language modeling research has been actively studied since 1980, and it is useful for a variety of applications, including speech recognition, machine translation, document classification, optical character recognition, information retrieval, handwriting recognition, and many more. A statistical language model (SLM) is a probability distribution over sequences of words that yields the relative probability of any given finite sequence. Assuming the word sequence is $W = w_1, w_2, ..., w_N$, the salient question for efficient language modeling is how to estimate $p(W)$. We could decompose $p(W)$ using the chain rule:

$$p(W) = p(w_1, w_2..., w_n) = p(w_1)p(w_2|w_1)p(w_3|w_1, w_2)...p(w_n|w_1, w_2, ...w_{n-1})$$

Then, the modeling task could be formulated as predicting the next word given the history of words, which is to estimate the conditional probability.

$$p(w_t|w_1, w_2, ..., w_{t-1})$$

The model then provides a probability distribution over all possible next words. The problem is often simplified by limiting the operating vocabulary to a fixed-size set of the most

frequently-used words. Words that are not in the vocabulary are replaced by a special word "<unk>".

Depending on the dataset, the vocabulary size can vary significantly. In real-world data sets, the vocabulary size is often multiplied due to spelling errors, alternate capitalization, and other variations. Language models are trained using representative corpora for the domain in questions. A corpus of, for example, bio-medical research literature will vary significantly in word content and frequency from a corpus obtained from legal texts or social media postings. Language models have been reported to be based upon corpora as large as 100 billion words [Shazeer et al., 2017].

Recurrent neural language (RNN) models (described in detail in Chapter 2) have been very effective and remained the state-of-the-art model in recent years. However, several challenges remain in constructing and training such models. Chief among these challenges is the computational complexity of model training as the number of parameters increases. Currently, the best performing recurrent neural models are those with the most parameters. On Google's one-billion-word corpus, the best model [Shazeer et al., 2017] has more than 4 billion parameters and is trained on 32 Tesla K40 GPUs. Thus, the best performing model for this corpus has more parameters than there are words in the training set. The demanding number of parameters for effective modeling makes it impossible to train on a single GPU to achieve the state-of-the-art performance. Assuming 32-bit values for each parameter, current state-of-the-art models require almost 15GB of memory just for parameter storage, and other 15GB to store parameter gradients during he training process. Given that the largest amount of storage currently available for off-the-shelf GPUs is 16GB, it would clearly be beneficial to reduce the parameter space if possible.

In addition to the extensive space complexity, training RNN language models is also quite time consuming, mainly because it requires estimating the softmax function at every time step. Many approaches have been proposed to reduce the time complexity of the training algorithm, such as hierarchical softmax [Goodman, 2001, Kim et al., 2016],

importance sampling (IS) [Bengio and Senécal, 2008, Jozefowicz et al., 2016], and noise contrastive estimation (NCE) [Mnih and Teh, 2012, Zoph et al., 2016]. However, it remains a significant challenge to efficiently train models as the number of parameters grows.

The development of more compact, more effective language models will allow for easier deployment in the real world, will reduce the communication overhead for training distributed models, and may even facilitate model use on mobile devices and other platforms with limited computational, power, and storage resources.

1.1 Evaluation of Language Models

Perplexity

A number of metrics are commonly employed to evaluate the performance of language modeling techniques. Among the simplest and most commonly reported is the perplexity (PPL). Given a sequence of words, W, the perplexity of the sequence relative to a given language model is defined as:

$$\text{PPL} = \sqrt[n]{\prod_{i=1}^{n} \frac{1}{P(w_i|w_{1\ldots i-1})}} \tag{1.1}$$

Perplexity can be understood as on average how many words with the same probability can appear given any context. With lower perplexity, the model has fewer words to choose from, so that it is more confident in its predictions, and the model is better. A closely related metric to PPL is cross entropy, which is simply the log of the perplexity.

BLEU Score

An important application of language modeling is machine translation. Unfortunately, perplexity is not a direct indicator of the performance of a trained model to machine trans-

lation. A common measure employed for this task is the BLEU score [Papineni et al., 2002]. BLEU score against a common corpus is typically used to measure the comparative performance of machine translation systems.

The machine translation system normally uses data comprising source language sentences, and their corresponding translated target language sentences. For each source language sentence, it can have more than one reference translated sentence. The BLEU score is defined as:

$$\text{BLEU} = \min(1, \exp(1 - \frac{\text{reference-length}}{\text{output-length}}))(\prod_{i=1}^{4} \text{precision}_i)^{\frac{1}{4}}$$

$$\text{Precision}_i = \frac{\text{clipped correct candidate i-gram}}{\text{total number of candidate i-gram}}$$

Here the clipped correct candidate i-gram means the maximum times the i-gram occurs in any single reference translation. The reference length considers the closest reference length for each translation. BLEU score is typically computed over the entire corpus, and is not defined on single sentences.

As a simple example, we have a simple test set that have one source sentence and two reference translated target sentences.

Candidate translation: Two dogs in the the apartment.

Reference 1: There are two dogs in the house.

Reference 2: Two dogs are in the room.

For unigram (1-gram), all the words occur in the reference sentences except ”apartment”, and ”the” occurs at most once. So clipped correct candidate 1-gram is 4. The total number of unigram in the candidate translation is 6. For bigram (2-gram), the clipped correct candidate 2-gram is 3 and the total number of bigram in the candidate translation is 5. For

trigram (3-gram), the clipped correct candidate 3-gram is 2. For 4-gram, the clipped correct candidate 4-gram is 1. The length of candidate translation is 6. And the closest reference length is 6. Thus, the BLEU score for this example is:

$$\min(1, \exp(1 - \frac{6}{6}))((4/6) * (3/5) * (2/4) * (1/3))^{0.25} = 0.508$$

We can see that the model with a higher BLEU score is better.

1.2 N-gram Language Model

The most successful language model in the first two decades of language modeling research was the n-gram language model [Rosenfeld, 2000]. The model is simple and works well when there is enough data. A major advantage of the n-gram language model is its interpretability.

The probability $P(w_t|w_{1,...,i-1})$ depends on the entire word history, and we can estimate it based on the counts that the word sequence occurs.

$$P(w_t|w_{1,...,i-1}) = \frac{\text{Count}(w_1, ..., w_t)}{\text{Count}(w_1, ..., w_{t-1})}$$

However, this is not computationally feasible. Almost all word sequences occur only a limited number of times, and it is impossible to record all the sequences. For example, if the vocabulary size is 10,000, and the total number of unique sequences with length 30 will be 10^{120}, it is larger than the number of atoms in the universe. Therefore, we need to make the Markov assumption that

$$P(w_t|w_{1,...,t-1}) \approx P(w_t|w_{t-1}, w_{t-n+1}) = \frac{\text{Count}(w_{t-n+1}, ..., w_t)}{\text{Count}(w_{t-n+1}, ..., w_{t-1})}$$

where the value n is a fixed number. This is the well-known n-gram e language model.

In order to overcome the data sparseness problem, various smoothing techniques [Chen and Goodman, 1999] have been proposed. The n-gram language model, which simply estimates the probability of occurrence of each word based on its relative frequency given previous n words in the training corpus, is very effective in practice when a larger n is used. In particular, 1) The model can be trained in a short time. It is easy to do distributed training using map/reduce techniques. 2) The model can be explained. If the model gives a bad prediction, we can find the reason. 3) It's fast and efficient at the decoding stage in speech recognition and machine translation, where only a look-up table is maintained. For these reasons, n-gram models are still widely used in the industry, despite its disability to take into account of long range dependency and the emergence of more accurate models over the past decade.

1.3 Benchmark Datasets

Several datasets have been used extensively in language modeling research. The most popular benchmark dataset is the Penn Treebank Corpus (PTB) [Marcus et al., 1993]. PTB consists of 929k training words, 73k validation words, and 82k test words. It has 10k words in its vocabulary. This data set is available for public download at the Linguistic Data Consortium website, and the processed data can be obtained from `http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz`. It has been found that when training RNN language models, overfitting occurs easily. Many regularization methods are tested in PTB dataset [Zaremba et al., 2014, Graves, 2013, Gal and Ghahramani, 2016].

Often, it has been found in practice that models that perform well on smaller datasets do not generalize well on larger ones. In 2013, Google released a public dataset [Chelba et al., 2014], that has almost one billion words. In addition to the large overall size of

this corpus, it is also distinguished by its sizable vocabulary, comprising 793471 words including start of sentence token "<s>" and end of sentence token "</s>". At the time of its release, state-of-the-art models were able to yield perplexity scores around 43.8 for this data. At the time of the writing of this dissertation, the best single-model perplexity that has been achieved for the Google data set is 28.0 [Shazeer et al., 2017].

Recurrent Neural Language Models

2.1 Simple Recurrent Neural Network

The state-of-the-art language model is currently Recurrent Neural Network (RNN) language models.

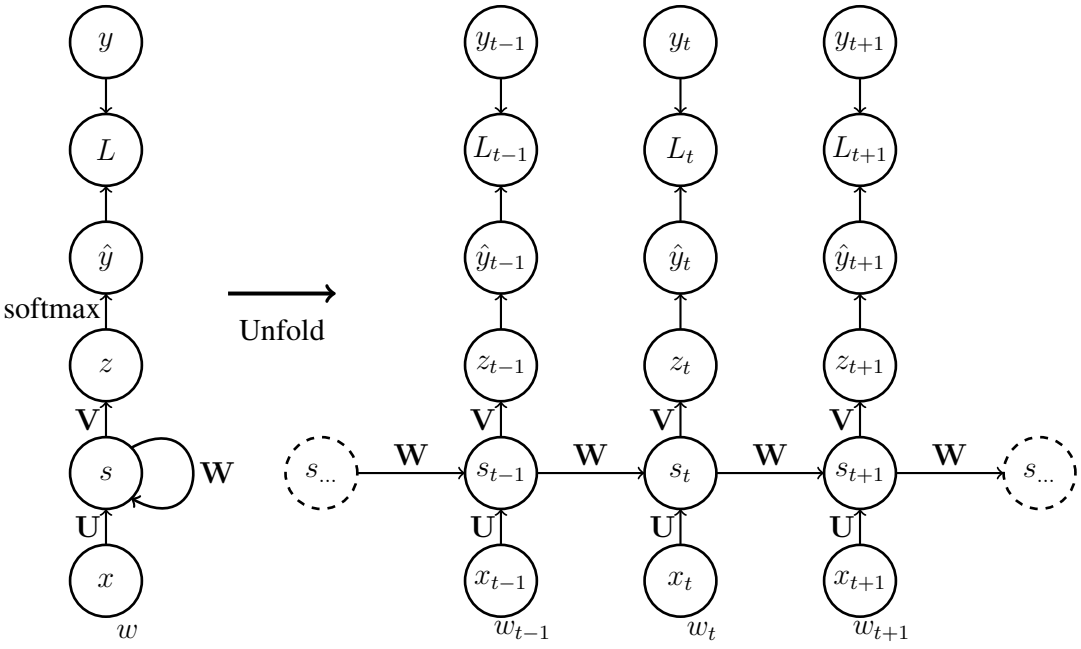

Figure 2.1: Simple Recurrent Neural Language Model

The left in Fig 2.1 is an RNN and its loss drawn with recurrent connections. The RNN has input to hidden connections parametrized by a weight matrix U, hidden-to-hidden recurrent connections parametrized by a weight matrix W, and hidden-to-output connections

parametrized by a weight matrix V. A loss L measures how far each $\hat{y}$ is from the corresponding training target y. The right in Fig 2.1 is the unfolded computational graph for a word sequence with length of T, $w_1, \ldots, w_T$, where each node is now associated with one particular time instance. The computational graph is used to compute the training loss of a recurrent network that maps an input sequence of x values to a corresponding sequence of output $\hat{y}$ values.

At each time stamp, the current word is encoded as one-hot vector. In most cases, the vocabulary of the language model is fixed and we assume the vocabulary size is V. Then each word's one-hot vector will have the size of V. For example, if the vocabulary size is 3, then the one-hot vector could be $[1, 0, 0]^T$, $[0, 1, 0]^T$, $[0, 0, 1]^T$. x_t is the current word w_t's one hot vector, and $s_t = f(\mathbf{U}x_t + \mathbf{W}s_{t-1})$. The matrix $\mathbf{U}$ is usually called *word embeddings*, because the matrix multiplication of the one hot vector with $\mathbf{U}$ is equivalent to just finding the row in which the one hot vector's location is also one. To compute the current timestep's hidden states s_t, the previous timestep's hidden state s_{t-1} is also used. f is a non-linear function, which usually is the sigmoid function

$$f(x) = \frac{1}{1 + \exp(-x)}.$$

The hidden state s_t is also called context vector. With s_t, it is projected to a vector of the same size as the vocabulary size using matrix $\mathbf{V}$, $z_t = \mathbf{V}s_t$ and then is fed into the softmax function to get a normalized output vector $\hat{y}_t$. So in the output matrix V, each word w also has one corresponding row vector v_w, called word embeddings , $z_t^w = v_w s_t$, then the element in $\hat{y}_t$ corresponding to word w is

$$\hat{y}_t^w = p(w|s_t) = \frac{\exp(z_t^w)}{\Sigma_{w'} \exp(z_t^{w'})}.$$

In language modeling, the loss L between each $\hat{y}_t$ and the corresponding training target

y_t is the cross-entropy,

$$L(\hat{y}_t, y_t) = -\sum_{i=1}^{V} y_{ti} \log(\hat{y_{ti}})$$

In the equation, and y_{ti} is ith value of the one hot vector for target word w_t, so that

$$\sum_{i=1}^{V} y_{ti} \log(\hat{y_{ti}}) = \log(p(w_t|s_t))$$

Thus the total loss of generating the whole sequence $\mathbf{w} = w_1, \cdots, w_T$ could be computed by the forward propagation described above as

$$\text{Loss}(\mathbf{w}) = -\log(\mathbf{w})) = -\sum_{t=1}^{T} \log(w_1, \cdots, w_T) = -\sum_{t=1}^{T}\sum_{i=1}^{V} y_{ti} \log(\hat{y_{ti}}) = -\sum_{t=1}^{T} \log(p(w_t|s_t))$$

So the loss is also called negative log-likelihood loss. It can be proved that minimizing this loss is equivalent to minimizing the KL-divergence between the data distribution and the model distribution.

2.2 Training of Simple Recurrent Neural Network

The dominate approach to train neural networks is the *minibatch stochastic gradient descent* (SGD) method [Goodfellow et al., 2016]. Instead of using the entire training set or only a single example at a time, minibatch is a good trade-off among several factors such as accuracy, efficiency, regularizing effect etc. In each round, we sample a minibatch of M examples from the training set, compute gradient estimate of loss function over these M examples, then apply gradient update for the model parameters. Repeat this procedure until convergence.

In language modeling, we keep the word order, and first concatenate all of the words of the entire corpus as one word sequence, we partition this sequence into M parts, where

M is the minibatch size and could be set to 64 for example and each part is treated as a word sequence, each part is independent with other parts, and then stack them together. So the total loss becomes

$$\frac{1}{M}\sum_{m=1}^{M}\text{Loss}(\mathbf{w}^m) = -\frac{1}{M}\sum_{m=1}^{M}\log(\mathbf{w}^m)$$

Each $\mathbf{w}_m, m = 1, \cdots, M$ is still a long sequence, again we partition each sequence into L subsequences, each with a length of N, the lth subsequence of $\mathbf{w}^m$ is $w^m_{(l*N)+1}, \cdots, w^m_{(l+1)*N}$ whose dependence with previous subsequence is only through the hidden state $s^m_{(l*N)}$. The loss could be rewritten as

$$\frac{1}{M}\sum_{m=1}^{M}\text{Loss}(\mathbf{w}^m) = -\frac{1}{M}\sum_{m=1}^{M}\log(\mathbf{w}^m) = -\frac{1}{M}\sum_{m=1}^{M}\sum_{l=0}^{L}\log(w^m_{(l*N)+1}, \cdots, w^m_{(l+1)*N}|s^m_{(l*N)})$$

and the loss for the subsequence of $\mathbf{w}^m$ is $w^m_{(l*N)+1}, \cdots, w^m_{(l+1)*N}$ can be computed by the forward propagation on the unrolled computational graph starting from the hidden state $s^m_{(l*N)}$.

Computing the gradient of the loss for the subsequence of $\mathbf{w}^m$ is $w^m_{(l*N)+1}, \cdots, w^m_{(l+1)*N}$ is straightforward too. One simply applies the generalized back-propagation algorithm to the unrolled computational graph. No specialized algorithms are necessary. This is what is most commonly called truncated back propagation through time (TBPTT) [Mikolov et al., 2011b, Jozefowicz et al., 2016, Werbos, 1990]. During back-propagation, we need to know the derivative of the loss function respect to the weights. Because of the chain rule, we will need to use the derivative of the activation functions. The derivative of the sigmoid function is:

$$\frac{\partial f(x)}{\partial x} = \frac{1}{1+\exp(-x)}(1 - \frac{1}{1+\exp(-x)}) = f(x)(1 - f(x))$$

For softmax function, the derivative is:

$$\begin{aligned}\frac{\partial p(w|s_t)}{\partial z_w} &= \frac{\exp(z_w)}{\sum_{w'}\exp(z_{w'})} - (\frac{\exp(z_w)}{\sum_{w'}\exp(z_{w'})})^2 \\ &= \frac{\exp(z_w)}{\sum_{w'}\exp(z_{w'})}(1 - \frac{\exp(z_w)}{\sum_{w'}\exp(z_{w'})}) \\ &= p(w|s_t)(1 - p(w|s_t))\end{aligned}$$

For $z_{w'}, w' \neq w$,

$$\frac{\partial p(w|s_t)}{\partial z_{w'}} = -\frac{\exp(z_w)}{\sum_{w'}\exp(z_{w'})}\frac{\exp(z'_w)}{\sum_{w'}\exp(z_{w'})} = -p(w|s_t)p(w'|s_t)$$

We can see that for sigmoid and softmax function, their derivative can be computed using the activation values, so that we do not need to evaluate the time consuming exponential function during back propagation.

As we know,

$$\log(w^m_{(l*N)+1}, \cdots, w^m_{(l+1)*N}|s^m_{(l*N)}) = \sum_{i=1}^{N}\log p(w^m_{(l*N)+i}|s^m_{(l*N)+i})$$

Let

$$Q_i(w) = -\log(p(w_i|s_i))$$

so that

$$\begin{aligned}\frac{\partial(Q_i(\theta))}{\partial z_w} &= \frac{\partial(Q_i(\theta))}{\partial p(w|s_t)}\frac{\partial p(w|s_t)}{\partial z_w} \\ &= -\frac{1}{p(w|s_t)}p(w|s_t)(1 - p(w|s_t)) \\ &= p(w|s_t) - 1.\end{aligned}$$

$$\begin{aligned}\frac{\partial(Q_i(\theta))}{\partial z_{w'}} &= \frac{\partial(Q_i(\theta))}{\partial p(w|s_t)}\frac{\partial p(w|s_t)}{\partial z_{w'}} \\ &= (-\frac{1}{p(w|s_t)})(-p(w|s_t)p(w'|s_t)) \\ &= p(w'|s_t)\end{aligned}$$

Once the gradient of the loss of the minibatch of subsequences are obtained, we update the parameters by gradient descent, then we move to the next minibatch of subsequences. When we reach the end of the sequences, we rotate back to the beginning of the M sequences. We repeated this procedure until convergence. Of course many variants of SGD exist, and Ruder [Ruder, 2016] gives a good survey.

2.3 Long Short-Term Memory

There are many different recurrent neural network architectures, but Long Short-Term Memory (LSTM) is the most commonly used. In sentences, long term dependencies exist. The next word we want to predict could be highly related to several words we mentioned many words away from the current position. For example, consider the following sentence:

"***SpaceX*** on Thursday launched and landed a first-stage ***rocket*** booster that had previously flown — a milestone that could signal a new era of low-cost ***space*** transportation".

In this sentence, the word "***space***" is highly related to the word "***rocket***" and "***SpaceX***". It is impossible to do this in normal n-gram language models, as the largest n-gram order I have found is 9-gram, and the n-gram baseline model for Google's one-billion-word dataset is 5-gram [Chelba et al., 2014]. However, it is possible to do this with the simple RNN language model. The key advantage of the recurrent neural network model is that each hidden state depends on all the words in the current history. While this enables modeling long-term dependencies, it also results in a network that is very difficult to train, due to the exploding

gradient or vanishing gradient problem [Bengio et al., 1994].

To resolve the difficulty in training simple RNNs due to vanishing gradients, LSTM introduces memory cells and does not use activation functions on the memory cells. The stored value in the cells could be kept for many timesteps. In the Wikipedia experiments of Graves [2013], the LSTM model trained was able to correctly model opening and closing quotation marks and parentheses, which is a clear indicator of the language model's memory. Since any sequence of words, including very long sequences, can be found between quotes or parentheses, it is impossible to correctly model them using only short-range context.

LSTM models have more parameters than simple RNN models with the same number of hidden states because of the gates. However, the work of Jozefowicz et al. [2016] shows that even with the same number of parameters, LSTM models perform better than the simple RNN models. LSTM has also been widely used in industry, for example it is used to generate email replies for Gmail [Google, 2015] and Apple also has its own version called "QuickType" [Wired, 2016].

In the experiments described here, we used LSTM as our basic building blocks of large language models. The internal dynamics of LSTM are shown in the following equations, in each timestep t, the word vector x_t is used as the input, then h_t is used to predict the next word y_t.

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} T_{2n,4n} \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} \tag{2.1}$$

$$c_t = f \odot c_{t-1} + i \odot g \tag{2.2}$$

$$h_t = o \odot \tanh(c_t) \tag{2.3}$$

In the formula, $\odot$ is element-wise multiplication, and $T_{n,m} : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is an affine transform, i is the input gate, f is the forget gate, o is the output gate, and g is the non-linearity for generating output of the current timestep. c is the memory cells. The gated design is very successful and is used to develop many new architectures, such as the Highway networks [Srivastava et al., 2015] and the gated convolutional language models [Miyamoto and Cho, 2016].

2.4 Stacking Recurrent Neural Networks

For language modeling, there is a large amount of training data. In the field of deep learning, models with larger numbers of parameters are trained more easily and generalize better [Jozefowicz et al., 2016]. One method to increase the number of parameters in the recurrent neural network is to increase the number of hidden states. The time complexity of LSTM layers is $\mathcal{O}(4N^2)$, when the dimension of the input vector x_t and the hidden state vector h_t are both N. But if we feed the output of the first LSTM layer to another layer and by stacking L layers of LSTM, the time complexity is $\mathcal{O}(4LN^2)$. And during training, the output layer is the most time consuming, so uses the most time. Therefore, by stacking more layers of LSTM [Graves, 2013], we can increase the number of parameters without significantly increasing the time complexity.

2.5 Regularization of Recurrent Neural Network

RNN language models contain a huge number of parameters, which makes it very easy to overfit. On Google's one-billion-word dataset [Chelba et al., 2014], overfitting has been observed even when training a LSTM model with 2048 hidden states. There are several methods that are used in recent published works to avoid overfitting in these models, such

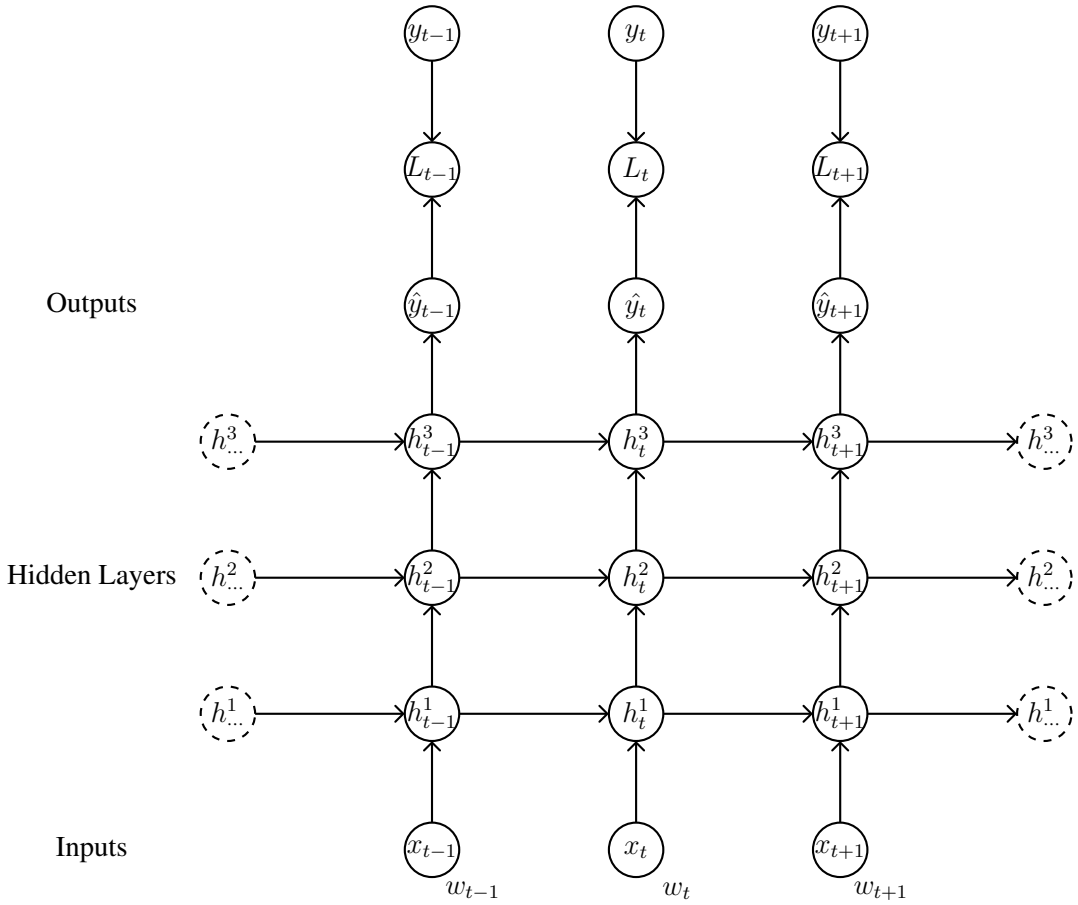

Figure 2.2: Deep recurrent neural network architecture. The circles represent network layers, the solid lines represent weighted connections.

as L2 regularization [Mikolov et al., 2010, Joulin et al., 2017], adding weight noise [Graves, 2013] and dropout[Zaremba et al., 2014].

Dropout

Dropout was the first highly successful method applied to regularize feed forward neural networks, and it remains widely popular today. It works by randomly dropping units and connections (setting hidden states to zero) from the neural network during training [Srivastava et al., 2014]. For recurrent neural networks, Zaremba et al. [Zaremba et al., 2014] found applying dropout to the non-recurrent layers is a very effective way to regularize the RNN model. For multiple layers of LSTM, the dropout works in the following way

Zaremba et al. [2014],

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} T_{2n,4n} \begin{pmatrix} \mathbf{D}(h_t^{l-1}) \\ h_{t-1}^l \end{pmatrix}$$

$$c_t^l = f \odot c_{t-1}^l + i \odot g$$

$$h_t^l = o \odot \tanh(c_t^l)$$

In the formula, **D** is the dropout operator that sets a random subset of its argument to zero.

L2 Regularization

L2 Regularization works by adding a penalty term to the loss function, it penalizes the large values. L2 regularization is very easy to implement and doesn't use extra space.

$$\mathcal{L}_{\text{L2}}(W) = \mathcal{L}(W) + \lambda W^2$$

The gradient is the following,

$$\frac{\partial \mathcal{L}_{\text{L2}}(W)}{\partial W} = \frac{\partial \mathcal{L}(W)}{\partial W} + 2\lambda W$$

The way to implement L2 regularization is to subtract a small, constant fraction of the current weight value at each training step.

Adding Noise

Dropout could be treated as a special case of adding activation noise to the network [Srivastava et al., 2014]. In the experiments of Graves [2013], two kinds of regularization were tested, one is adding noise with standard deviation of 0.075 to the network weights at the beginning of each sequence, and the other is adding adaptive noise. For the word level experiments, weight noise was used because computation of the adaptive noise was too slow. For the PTB dataset, the word level model trained with weight noise yielded the best PPL.

2.6 Noise Contrastive Estimation

Recall that language models attempt to predict the probability of the next word in a sequence, given the current word history. Since this probability distribution must be calculated over (and normalized by) all words in a language, the normalization step can be computationally prohibitive. Several sampling based approaches to speeding up this step have been proposed, including noise contrastive estimation and importance sampling (NCE) [Gutmann and Hyvärinen, 2010, Zoph et al., 2016, Dyer, 2014]. NCE reduces the language model estimation problem to a binary classification problem, the problem of estimating the classifier that uses the same parameters to distinguish samples from the empirical distribution from samples generated by the noise distribution. For language modeling, the data distribution $p_d^h(w)$ depends on the context, which is the last hidden state vector in the current timestep. The noise distribution $p_n(w)$ is known and fixed. For language modeling, the noise distributions usually used are the uniform distribution, the unigram distribution or flattened unigram distribution (unigram distribution power raised to some value and then normalized). And it is also possible to use context-dependent distributions. NCE makes the normalization term as an additional parameter.

The training data set D is generated with the following method, for each context h, we first sample one "true" sample for the empirical distribution, and use an auxiliary label

$d = 1$ to indicate this data point is drawn from the true distribution. Then we sample k "noise" samples from $p_n(w)$, with auxiliary label $d = 0$ indicating these data points are noise. Therefore the joint distribution of label and data is

$$p(d, w|h) = \begin{cases} \frac{k}{k+1} \times p_n(w) & \text{if} \quad d = 0 \\ \frac{1}{k+1} \times p_d^h(w) & \text{if} \quad d = 1 \end{cases}$$

Using the formula of conditional probability, we can find the probability of the label d given the context h and observed word w:

$$\begin{aligned} p(d = 0|h, w) &= \frac{\frac{k}{k+1} \times p_n(w)}{\frac{1}{1+k} \times p(w|h, w) + \frac{k}{1+k} \times p_n(w)} \\ &= \frac{k \times p_n(w)}{p(w|h) + k \times p_n(w)} \end{aligned}$$

$$\begin{aligned} p(d = 1|h, w) &= \frac{\frac{1}{k+1} \times p(w|h)}{\frac{1}{1+k} \times p(w|h, w) + \frac{k}{1+k} \times p_n(w)} \\ &= \frac{p(w|h)}{p(w|h) + k \times p_n(w)} \end{aligned}$$

In the above formulas, the probability $P(w|h)$ still needs to be normalized. Let $u(w|h) = \exp(z_w)$. NCE makes two further assumptions. First, the partition function $Z(h) = \sum_w \exp(z_w)$ could be estimated as parameters, which means for every history, NCE introduces one parameter. Second, for neural networks with numerous parameters, fixing $Z(h) = 1$ for all h is effective [Mnih and Teh, 2012].

$$p(d = 0|h, w) = \frac{k \times p_n(w)}{u(w|h) + k \times p_n(w)}$$

$$p(d=1|h,w) = \frac{u(w|h)}{u(w|h) + k \times p_n(w)}$$

Now the problem is changed to be a binary classification problem that can be trained to maximize conditional log-likelihood of d, with k negative samples chosen:

$$\mathcal{L}_{\mathrm{NCE}_k} = -\sum_{(w,h)\in D} (\log p(d=1|h,w) + kE_{\hat{w}\sim p_n(w)} \log p(d=0|h,\hat{w}))$$

The second term of the above function is still a summation over the whole vocabulary, thus it is still too expensive to calculate. The final step is to replace the expectation with a Monte Carlo approximation:

$$\begin{aligned}\mathcal{L}^{\mathrm{MC}}_{\mathrm{NCE}_k} &= -\sum_{(w,h)\in D} (\log p(d=1|c,h) + k \times \sum_{i=1,\hat{w}\sim p_n(w)}^{k} \frac{1}{k} \times \log p(d=0|h,\hat{w})) \\ &= -\sum_{(w,h)\in D} (\log p(d=1|h,w) + \sum_{i=1,\hat{w}\sim p_n(w)}^{k} \log p(d=0|h,\hat{w}))\end{aligned}$$

There are several requirements for the noise distribution to become a good candidate. 1) The analytical expression for $p_n(w)$ is known. 2) A noise distribution must be sampled easily. 3) A noise distribution that in some aspect, for example with respect to covariance structure, is similar to the data must be chosen. 4) The number of noise samples needs to be as large as possible. These requirements could be useful when improving the performance of the models.

2.7 Importance Sampling

In the softmax layer, the context vector s_t needs to be projected to a vector with the same size of the vocabulary size. If the dimension of the hidden vector is H, then the complexity of the softmax layer will be VH. When the vocabulary of the dataset is large, this matrix multiplication is very time consuming. Another approach to solve this problem is to use importance sampling (IS) [Bengio and Senécal, 2008, Jozefowicz et al., 2016]. For the full softmax function and its derivative:

$$p(w|s_t) = \frac{\exp(z_w)}{\sum_{w'} \exp(z_{w'})}$$

$$\begin{aligned}
\frac{\partial(-\log(p(w|s_t)))}{\partial\theta} &= -\frac{\partial z_w}{\partial\theta} + \sum_{w'}(\frac{\exp(z_{w'})}{\sum_{w'}\exp(z_{w'})}\frac{\partial z_{w'}}{\partial\theta}) \\
&= -\frac{\partial z_w}{\partial\theta} + \sum_{w'} p(w')\frac{\partial z_{w'}}{\partial\theta} \\
&= -\frac{\partial z_w}{\partial\theta} + E_P[\frac{\partial z_{w'}}{\partial\theta}]
\end{aligned}$$

Importance sampling is very straightforward. In order to approximate the average $E_P[\frac{\partial z_{w'}}{\partial\theta}]$, we can use a proposal distribution $Q(w)$.

$$\sum_{w'} p(w'|s_t)\frac{\partial z_{w'}}{\partial\theta} = \sum_{w'} Q(w')\frac{p(w')}{Q(w')}\frac{\partial z_{w'}}{\partial\theta}$$

Then, we apply classic Monte-Carlo to estimate the gradient. We take m independent samples $y_1, ..., y_m$ from $Q(w')$, then the estimator is:

$$\frac{1}{m}\sum_{i=1}^{m}\frac{p(y_i|s_t)}{Q(y_i)}\frac{\partial z_{y_i}}{\partial\theta}.$$

As explained in Bengio and Senécal [2008], in order to compute $p(y_i|s_t)$, the partition function $\sum_{w'} \exp(z_{w'})$ still needs to be computed. It was suggested that when we approximate the expectation, we use

$$p(w|s_t) \approx p'(w|s_t) = \frac{\frac{\exp(z_w)}{Q(w)}}{\sum_{w' \in [y_1, y_2, ..., y_m]} \frac{\exp(z_{w'})}{Q(w')}}$$
$$= \frac{\exp(z_w - \log Q(w))}{\sum_{w' \in [y_1, y_2, ..., y_m]} \exp(z_{w'} - \log Q(w'))} \text{ when } w \in [y_1, ..., y_n]$$

$$\frac{\partial(-\log(p(w|s_t)))}{\partial\theta} = -\frac{\partial z_w}{\partial\theta} + E_P[\frac{\partial z_{w'}}{\partial\theta}]$$
$$\approx -\frac{\partial z_w}{\partial\theta} + \sum_{w' \in [y_1, y_2, ..., y_n]} p'(w'|s_t)\frac{\partial z_{w'}}{\partial\theta}$$
$$= -\frac{\partial(z_w - Q(w))}{\partial\theta} + \sum_{w' \in [y_1, y_2, ..., y_n]} p'(w'|s_t)\frac{\partial(z_{w'} - \log Q(w')}{\partial\theta}$$
$$= \frac{\partial(-\log(p'(w|s_t)))}{\partial\theta}$$

This also makes the implementation very straightforward, we simply subtract the $\log Q(w)$ from each sampled z_w, and then pass them to the softmax layer (already implemented in many frameworks), then to the loss which predicts the target, and after the backward pass, we get the gradient with respect to z_w.

The time complexity of IS is mH, because in each time stamp, we just need to evaluate m samples' terms. The approximate gradient could be used to train the language model.

2.8 Class-based Softmax Layer

The computation bottleneck of neural language model is in the output layer, where there is a dense matrix multiplication with the complexity that depends on the vocabulary size

of the corpus. Aside from the above two sampling based methods, a simple and popular technique to reduce the complexity is to use a class based output layer [Mikolov et al., 2011a]. In this approach, we assign each word a fixed class id. For example if we have C classes in total, any word in the vocabulary has one unique class id.

$$p(w|h) = p(c(w)|h)p(w|c(w), h)$$

When we try to compute the probability of a word, first we compute the probability of the class, then compute the probability of the word within the class. In this way we no longer need to compute the softmax over the full vocabulary. If we assume the number of words in each class is almost equal, then the time complexity of the model is

$$HC + \frac{HV}{C}$$

It is easy to see that when $C = \sqrt{V}$, then time complexity is the smallest $2H\sqrt{V}$.

There are many ways to assign the class ids to the words. The simplest method is to simply randomly partition the words into C sets [Kim et al., 2016]. Another way is to do the assignment according to the frequency of the words. If the class set size is not restricted to be equal, then we see that we can assign the most frequent words in a smaller set, which also helps to reduce the time complexity.

Another way to look at this model is that we can treat it as a two-level hierarchical softmax approximation [Morin and Bengio, 2005]. In hierarchical softmax approximation, each word is a leaf node, and the probability of the word is the probability of the path from the root node to the corresponding leaf node. If we assume all the words are in the leaves and the tree is a complete tree, then the depth of the tree is $\log(V)$, and the complexity for computing the probability of a word is also $O(\log(V))$.

It is clear to see that the training complexity is decreased to $O(\log(N))$. But for

inference, if we want to find the word with the exact largest probability, the complexity is still $O(N)$, because we need to compute the probability of all the words. One way to speed this up is to use some way to shortlist the possible words and only score the most probable list of words. This kind of model is used in Google's chatting app Allo [Google, 2016].

An interesting observation for class-based softmax and hierarchical softmax is that they all increase the space complexity of the model when the same dimension of hidden states is used.

2.9 Training Best Practices

The following section summarizes current best practices in training neural network based language models.

2.9.1 Adding Dropout

As previously noted, deep learning models have a large number of parameters to train, in some cases even exceeding the size of the data set used for training. The most commonly employed and successful technique to mitigate overfitting of parameters is to employ dropout. In some cases, the number of parameters is even larger than the size of the dataset [Jozefowicz et al., 2016]. For LSTM language models, dropout can only be used in the non-recurrent connections. On one-billion-word dataset, a language model with 2048 hidden states still has overfitting.

2.9.2 Mini-batch Training

The use of graphics processing units (GPUs), which are optimized for large matrix multiplication operations, has facilitated the training of much larger networks than were previously feasible. Mini-batch training, where the size of the batch is optimized to efficiently use the

GPU resources, results in significant performance increases.

2.9.3 Fix Upper Bound of Gradient

Recurrent neural networks are known to suffer from gradient decay or gradient exploding. Experiments show that fixing the upper bound of the gradient helps to solve the gradient exploding problem [Pascanu et al., 2013]. Normally the upper bound of the gradient is set to be 0.5 to 10 times the average gradient norm.

2.9.4 Use Adagrad

The Adagrad algorithm [Duchi et al., 2011] is easy to implement and is found to be very successful when training on the one-billion-word dataset. The update rule for SGD algorithm is:

$$W_{t+1,i} = W_{t,i} - \eta g_{t,i}$$

The update rule for Adagrad algorithm is:

$$W_{t+1,i} = W_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i}$$

where G_t is a diagonal matrix where each diagonal element is the sum of the squares of the gradients w.r.t. W_i up to time step t.

2.9.5 Sharing Noise Samples Across Mini-batch

In noise contrastive estimation and importance sampling training processes, it is necessary to have noise samples for every training example. If we share the noise samples for each mini-batch, the noise samples' embedding could be multiplied with all the hidden states in the mini-batch together, which also improves the training speed. Also, experiments

show that this does not negatively affect the performance of the result [Zoph et al., 2016, Jozefowicz et al., 2016].

2.9.6 Tuning Learning Rate

Although the Adagrad algorithm makes it easier to tune the learning rate, we have found that the learning rate still affects the convergence speed. It is helpful to first tune the learning rate using a smaller model, and then use the same learning rate for the larger models.

2.10 Parameter Sharing

If we have a model $F = f(W_1X_1 + W_2X_2)$, a simple way to understand parameter sharing is to let $W_1 = W_2$, and make the model be $F = f(WX_1 + WX_2)$. The gradient respect to W is the sum of the gradient respect to W_1 and W_2. It is easy to prove this.

$$\frac{\partial F}{\partial W} = \frac{\partial F}{\partial W}(X_1 + X_2)$$

$$\frac{\partial F}{\partial W_1} = \frac{\partial F}{\partial W_1}(X_1)$$

$$\frac{\partial F}{\partial W_2} = \frac{\partial F}{\partial W_2}(X_2)$$

If we let $W_1 = W_2 = W$, then

$$\frac{\partial F}{\partial W} = \frac{\partial F}{\partial W_1} + \frac{\partial F}{W_2}$$

Parameter sharing is widely used to reduce the number of parameters in neural network models. The most widely used model is convolutional neural network (CNN) [LeCun et al., 1998]. In CNN, the core building block is the convolutional layer, which uses a num-

ber of filters to apply convolution operation. For the forward computation, each filter is convoluted across the height and width of the image, and the activation is the dot product between the weights of the filter and the specific region of the image. The exact output of the layer can be described as:

$$\text{output}_{ijk} = \text{bias}_k + \sum_{l} \sum_{s=1}^{kW} \sum_{t=1}^{kH} \text{weight}_{s,t,l,k} * \text{input}_{dW(i-1)+s, dH*(j-1)+t, l}$$

If the input image is a 3D tensor nInputPlane $\times$ height $\times$ width, the output image size will be nOutputPlane $\times$ oheight $\times$ width where

$$\text{owidth} = \text{floor}((\text{width} + 2 * \text{padW} - kW)/dW + 1)$$

$$\text{oheight} = \text{floor}((\text{height} + 2 * \text{padH} - kH)/dH + 1)$$

And kW is the filter width, kH is the filter height, dW is the step of the convolution in the width dimension, dH is the step of convolution in the height dimension. padW is additional zeros added to the input plane data on both sides of width axis. padH is additional zeros added to the input plane data on both sides of height axis.

The total number of parameters is nOutputPlane $\times$ nInputPlane $\times$ kH $\times$ kW. We can see that the number of parameters in this layer doesn't depend on the width or height of the image. For different regions, the weights of the filter are shared. Otherwise, when dealing with large images, the model size could be too large to be handled.

For language modeling, Press and Wolf [2017] proposes to make the input layer and output layer share the same word embedding parameters, which helps improves the state-of-the-art language model performance. In a normal neural language model, each word has one input layer word embedding and one output layer word embedding. The method is to let them share the embedding.

2.11 HashNet

In machine learning, when dealing with a large number of features, the feature hashing is a common technique. If the features are sparse and the number is too large, it is not possible to give each feature one weight parameter. Feature hashing involves using some hashing function to hash the features into a fixed size buckets, the features that fall into the same bucket share the same weight.

One example of using the hashing trick in a neural language model can be seen in the work done by Mikolov et al. 2011b. In the proposed RNNME model, it hashes context n-gram input into the buckets, and uses the bucket weight as additional input to the model. The number n-gram's in a corpus could be huge, and it is impossible to assign one weight to each n-gram.

Chen et al. 2016 proposes to compress neural network models with the hashing trick. They propose using a low-cost hash function to hash the weight matrix positions into different positions in a fixed bucket array. It randomly group the weights of the neural network into different buckets, and the weights in the same bucket share the same weight. Another way to understand HashNet is that it is doing random parameter sharing.

2.12 Space Complexity Analysis of RNN Model

GPU memory usage during neural network training mainly has two parts: 1) Memory used to store the hidden states. 2) Memory used to store the parameters and their gradients; The space required by the parameters is normally equivalent to that needed to store the gradients. For a normal language model using the same dimension of embedding and hidden states H, with vocabulary size of V, the space needed to store the hidden states depend on the batch size B and BPTT length L. The bottleneck for storing the hidden states is on the output layer, because for each predicted word on current batch, it is necessary to

store a V size vector, taking $B * L * V$ space. In the example listed in the blog, just the output layer takes up to 20 Gb of memory! For the parameters and their gradients, the space bottleneck is on the embedding layers (including the input embedding layer and the output embedding layer). The embedding layer's size depends on the vocabulary size, and the space taken by the embedding layers is VH. The space usage of the parameters does not depend on the batch size or BPTT length, because the parameters are shared across different timesteps, and the gradients in the mini-batch are accumulated in the same place.

Sampling based softmax and hierarchical softmax layer help reduce the memory usage of storing the hidden states. Parameter sharing methods and adding projection layers help reduce the memory usage of parameters and their gradients. The simplest method is to make the input layer and output layer share the same word embedding [Press and Wolf, 2017], which cuts the space usage of parameters in half. Using a smaller dimension size of embedding is another simple way to reduce the memory usage of parameters.

HashNet [Chen et al., 2016] uses random parameter sharing to reduce memory usage. However, because the memory access pattern of GPU, it is necessary to construct the full parameter matrix before the computation. In the same mini-batch, we only construct the parameter matrix once.

The workload of inference is very different from training. During inference, it is not necessary to store all the hidden states intended for BPTT. During training, sampling based softmax doesn't evaluate the probability for all the vocabulary. However, for inference, if we want to get the exact prediction probability, we need to evaluate the full softmax. For rescoring, the dot product between the hidden states and the output layer embedding can be used as the score instead of the exact probability [Williams et al., 2015]. This not only reduces the memory usage during inference, but also make inference even faster than class based softmax or hierarchical softmax.

It has been demonstrated that 32 bits float computation is not needed for inference. Fixed point quantization can be used to transform the floating number computation into 8

bits integer computation. This technique is usable in all neural network models, and can be combined with the above mentioned methods. The usage of trained neural network models is limited to inference, so improving the inference efficiency is very important for large scale services. Google has even developed specialized hardware to improve the inference efficiency.

2.13 Related Work

There are many efforts to improve the space efficiency of neural language models. Kim et al. [2016] works with character level input and combines convolution neural networks (CNN) with highway networks to reduce the number of parameters. Later Jozefowicz et al. [2016] extends the CNN embedding idea to the output layer, comes up with a new CNN softmax layer, and scales the method to a one-billion-word corpus [Chelba et al., 2014]. Ling et al. [2015] introduces a model for constructing vector representations of words by composing characters using bidirectional LSTMs.

Although the above models use the character level information to reduce the model size of embedding layers, there have been many approaches that try to reduce the parameters without using this additional information. Mikolov et al. [2011a] introduces the compression layer between the recurrent layer and the output layer. This method not only reduces the number of parameters in the output layer, but also reduces the time complexity of training and inference. Joulin et al. [2017] improves the hierarchical softmax by assigning word clusters with different sizes of embeddings, it tries to utilize the power of GPU computation more efficiently, but also reduces the number of parameters significantly.

Chen et al. [2016] proposes to represent rare words by sparse linear combinations of common already learned word embeddings. The sparse code and embedding for each word are precomputed and fixed during the language model training process. The method we propose here differs in that the codes for each word are selected randomly and the embed-

dings are learned in the process of model training, and the sub-vectors are concatenated together to form the final word embedding.

Li et al. [2016] uses 2-component shared embedding as the word representation in LightRNN. It uses parameter sharing to reduce the model size. This is similar to our method. However, the two components for a word are fed into RNN in two different timesteps in LightRNN. The method proposed here is most similar to the model used in Suzuki and Nagata [2016], but the work is not about language modeling.

The model proposed here introduces random weight sharing into the embedding layers for language models similar to HashNet [Chen et al., 2015], but uses a different sharing scheme.

2.14 Problem Statement

Neural language models have been the state-of-the-art model for language modeling. These models encode words as vectors (word embeddings) and then feed them into the neural network [Bengio et al., 2003, Mikolov et al., 2010]. The word vectors are normally trained together with the model training process. In the output layer, the hidden states are projected to a vector with the same size as the vocabulary; Then a softmax function translates the vector into probabilities.

Training neural language models is time consuming, mainly because it requires estimating the softmax function at every time stamp. There have been many efforts that try to reduce the time complexity of the training algorithm, such as hierarchical softmax [Goodman, 2001, Kim et al., 2016], importance sampling (IS) [Bengio and Senécal, 2008, Jozefowicz et al., 2016], and noise contrastive estimation (NCE) [Mnih and Teh, 2012, Zoph et al., 2016]. It is also desirable to train very compact language models for several reasons: 1) Smaller models are easier to use and deploy in real world systems. If the model is too large, it is possible that it will need multiple server nodes. 2) Mobile devices have limited

memory and space, which makes it impossible to use large models without server access. 3) Smaller models also decrease the communication overhead of distributed training of the models.

There is significant redundancy in the parametrization of deep learning models [Denil et al., 2013]. Various pruning and parameter reduction methods are proposed. In general, there are two types of neural network compression techniques. The first type involves retraining. First a full-size model needs to be trained and its weights are pruned. Then the model is retrained [Han et al., 2016, See et al., 2016]. The second type involves encoding parameter sharing into the model and directly train the compressed model, such as HashNet [Chen et al., 2015]. The approach proposed here is the second type.

The input layer and output layer contain the largest portion of parameters in neural language models since the number is dependent on the vocabulary size. We mainly focus on reducing the number of parameters in the embedding layers. The main contribution is introducing a simple space efficient model compression method that randomly shares structured parameters, and can be used in both the input layer and the output layer. The method is easy to implement and tune. It can also be viewed as a regularization that leads to improved performance on perplexity and BLEU scores in certain cases.

2.15 Major Contributions

2.15.1 Space Complexity Reduction

We proposed a novel sub-vector based random parameter sharing method that can significantly reduce the space usage of language model.

We obtained good perplexity on datasets from small, medium to large (1 Billion) while using only a small fraction of the parameters required by other comparable approaches..

2.15.2 Inference Time Complexity Reduction

Based on dynamic programming, the time complexity of evaluating output layer is reduced, which provides faster inference speed than the full softmax layer both on CPU and GPU.

Our proposed method also has much better cache locality than HashNet, which helps our method achieve better inference speed.

The proposed methods help decrease the number of GPUs needed to train language models when the vocabulary size is large.

Slim Embedding Layers for Recurrent Neural Language Models

3.1 Random Parameter Sharing at Input and Output Embedding Layers

We used deep Long Short-Term Memory (LSTM) as our neural language model. In each time stamp t, the word vector h_t^0 is used as the input. We use subscripts to denote time stamps and superscripts to denote layers. Assume L is the number of layers in the deep LSTM neural language model, then h_t^L is used to predict the next word y_t. The dynamics of an LSTM cell, following Zaremba et al. [2014], are implemented as shown in equations 2.2 - 2.4.

Assuming the vocabulary size is V, and both the word vector size and the number of hidden nodes in the recurrent hidden states are N, then the total number of parameters in the embedding layer is $N * V$. The embedding layers of character level models [Kim et al., 2016, Miyamoto and Cho, 2016, Ling et al., 2015] are related in that the word embeddings between different words are dependent on each other. Updating the word embedding for each word will affect the embeddings for other words. Dependent word embedding helps reduce the number of parameters tremendously. We designed a simple model compression

method that allows the input word embedding layer and softmax output layer to share weights randomly to effectively reduce the model size and yet maintain the performance.

3.1.1 Compressing Input Embedding Layer

Assume we divided the input word embedding vector $x_t \in \mathbb{R}^N$ into K even parts, such that the input representation of the current word is the concatenation of the K parts $x_t = [a_1, ..., a_K]$, and each part is a sub-vector with $\frac{N}{K}$ parameters. For a vocabulary of V words, the input word embedding matrix thus is divided into $V * K$ sub-vectors, and we map these sub-vectors into M sub-vectors randomly but as uniformly as possible in the following manner: we initialize a list L with $K * V$ elements, which contains $\frac{K*V}{M}$ copies of the sequence $[1...M]$. Then the list is shuffled with the Fisher-Yates shuffle algorithm [Fisher et al., 1957] and the ith word's vector is formed with $[a_{L_{K*(i-1)+1}}...a_{L_{K*i}}]$. This helps to make sure that the number of times each sub-vector is used is nearly equal.

In this way, the total number of parameters in the input embedding layer is $M * \frac{N}{K}$ instead of $V * N$, which makes the number of parameters independent from the size of vocabulary. The K sub-vectors for each word are drawn randomly from the set of M sub-vectors.

For example, as shown in Fig 3.1, if there are four words total in the corpus (V=4), each word vector is formed by two sub-vectors (K=2), and there are in total eight sub-vectors in the input embedding matrix, assume that these eight sub-vectors are mapped into three sub-vectors (M=3), which are indexed as $a_i, i \in (1, 2, 3)$. Then the word vectors can be assigned like this: $[a_1, a_2], [a_1, a_3], [a_2, a_3], [a_3, a_1]$. In this example, the compression ratio is $3/8$, and the number of parameters in the new embedding layer size is only 37.5% of the original one.

If the number of sub-vectors is large enough and none of the word vectors share sub-vectors, then the input embeddings will become equivalent to normal word embeddings.

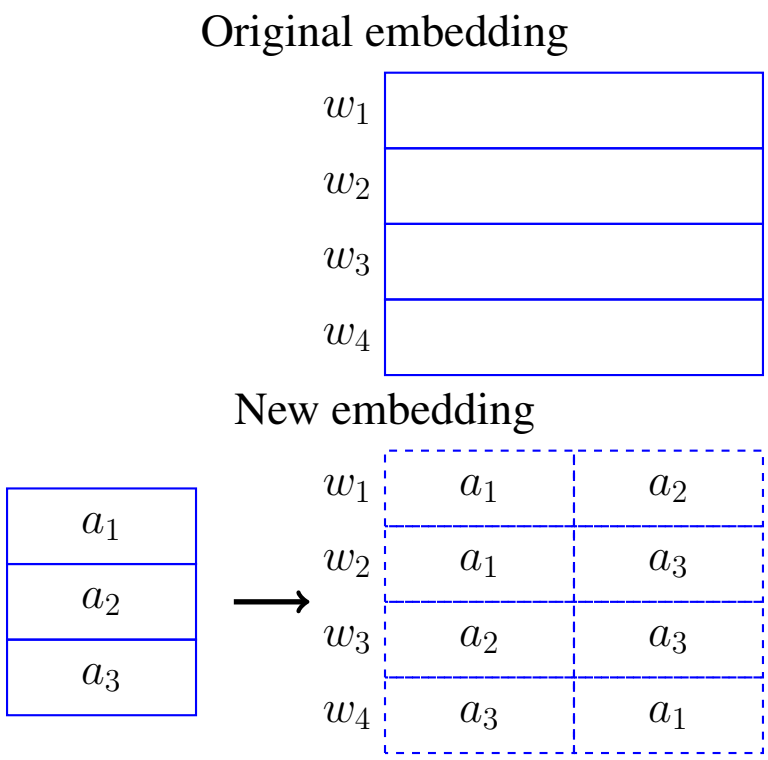

Figure 3.1: Toy example of the original embedding layer and new embedding layer. In this dissertation, the concatenated word vector has the same size as the original one. The assignment of sub-vectors to each word are randomly selected and fixed before the training process.

We used stochastic gradient descent with backpropagation through time [Werbos, 1990] to train our compressed neural language model. During each iteration of the training process, all words that shared the same sub-vectors with the current word were affected. If we assume that the number of words that share sub-vectors is small, then only a small number of word embedding vectors will be affected.

3.1.2 Compressing Output Embedding Layer

The output matrix can be compressed in a similar way. In the output layer, the context vector h is projected to a vector with the same size as the vocabulary, such that for each word w, we compute $z_w = h^T e_w$, which is then normalized by a softmax non-linearity: $p(w) = \frac{\exp(z_w)}{\Sigma_{w' \in V} \exp(z_{w'})}$. If we treat each e_w as a word embedding, we can then use a similar parameter sharing technique to the one used in the input layer, and let $e_w = [a_{w1}, ..., a_{wK}]$ where a_i are sub-vectors.

The structured shared parameters in the output layer make it possible to speed up the computation during both training and inference. Let S be K sets of sub-vectors, $S_1, S_2, ..., S_K$, such that $S_i \cap S_j = \emptyset, \forall i \neq j$. The first sub-vector in each word's embedding will be selected from S_1, the second from S_2, and so on. If we also divide the context vector as K even parts $h = [h_1, ..., h_K]$, then $z_w = \Sigma_{i=1}^{i=K} h_i^T a_{wi}$. We can see that h_i will only be multiplied by the sub-vectors in S_i. Because many words share the same sub-vectors, for each unique $h_i a_{wi}$, we only need to compute the partial dot product once. In order to evaluate all z_w, we need two steps with dynamic programming:

1) We first compute all the unique $h_i a_{wi}$ values. It is easy to see that the total number of unique dot product expressions will be the same as the total number of sub-vectors. The complexity of this step is $O(\frac{MH}{K})$, where M is the total number of sub-vectors. This step can be done with K dense matrix multiplications.

2) Each z_w is the sum of K partial dot products. Because the dot product results are already known from the first step, all we need to do is find the sum of the K values for each

word. The complexity of this step is $O(VK)$.

In summary, the complexity of evaluating the new softmax layer will be $O(\frac{MH}{K}+VK)$, instead of $O(VH)$ for the original softmax layer. The inference algorithm is listed in Algorithm 1.

1 Divide the hidden vector h into K even parts;
2 Evaluate the partial dot products for each (hidden state sub-vector, embedding) pair and cache the results;
3 Add the result for each word together according to the sub-vector mapping table;

Algorithm 1: Inference Algorithm

A simpler way to understand this algorithm is through an example. Assume we have four words, and four unique sub-vectors. The sub-vector assignment is like below,

$$W_1 = [a_1, a_3]$$

$$W_2 = [a_2, a_4]$$

$$W_3 = [a_1, a_4]$$

$$W_4 = [a_2, a_3]$$

Then to compute the softmax we need to compute

$$Z_1 = a_1h_1 + a_3h_2$$

$$Z_2 = a_2h_1 + a_4h_2$$

$$Z_3 = a_1h_1 + a_4h_2$$

$$Z_4 = a_2h_1 + a_3h_2$$

For the same color partial dot product, we only need to compute once and then cache the

result.

3.2 Connection to HashNet, LightRNN and Character Aware Language Model

The most similar work to our method is the HashNet described in Chen et al. [2015]. In HashNet, all elements in a parameter matrix are mapped into a vector through a hash function. However, in our approach, we randomly share sub-vectors instead of single elements. There are three advantages in our approach, 1) Our method is more cache friendly: since the elements of the sub-vectors are adjacent, it is very likely that they will be in the same cache line, thus it accesses the memory more efficiently than HashNet. Although the sub-vectors are randomly selected, the sub-vectors help improve the memory locality, thus speeding up the training and inference of the model. 2) Our method actually decreases the memory usage during training. When training Hashnet on GPU, the parameter mapping is usually cached, thus actually saving no space. 3) As shown in the previous section, it is possible to use dynamic programming to reduce the time complexity of the output layer with a simple modification. And if the sub-vector's size is equal to 1, the method proposed becomes one kind of HashNet, that uses a different hash function.

Our approach differs from LightRNN [Li et al., 2016] in that our approach is able to control the compression ratio to any arbitrary value. LightRNN can only compress at the rate of square or cube root of vocabulary size, which could be too harsh in practical applications.

The character-aware language model can be explained as a parameter sharing word-level language model. Each word shares the same character embedding vectors and a convolutional neural network (CNN). This model can also be explained as a simplified character-aware language model from Kim et al. [2016], Jozefowicz et al. [2016]. In the

character-aware language model, each character in a word is first encoded as a character embedding. It then uses a CNN to extract character n-gram features. These features are then concatenated and fed through several layers of highway network to form the final word embedding. In this model, if we treat the sequence of sub-vector ids (virtual characters) as each word's representation, the word embedding can be treated as concatenated unigram character feature vectors. The advantage of using the real character representation is that it can deal with out-of-vocabulary words nicely. However, the cost is that the model is more complicated and it needs to precompute the word embeddings for the words to speed up inference, so it couldn't stay in its compact form during inference. The model proposed here is much simpler, and easier to tune. During inference, it uses much less space and could even decrease the complexity of inference. With the same space constraint, this will enable us to train language models with even larger number of hidden states.

3.3 Experiments

We tested our method of compressing the embedding layers on various publicly available standard language model data sets ranging from the smallest corpus, PTB [Marcus et al., 1993], to Google's BillionW [Chelba et al., 2014] corpus. 44M is the 44 million word subset of the English Gigaword corpus [Graff and Cieri, 2003] used in Tan et al. [2012]. The descriptions of the datasets are listed in Table 3.1.

The weights are initialized with uniform random values between -0.05 and 0.05. Mini-batch stochastic gradient decent (SGD) is used to train the models. For all the datasets except the 44M corpus, all the non-recurrent layers except the word embedding layer to the LSTM layer use dropout. Adding dropout did not improve the results for 44M and BillionW, so the no-dropout results are shown. We used Torch [Collobert et al., 2011] to implement the models, and the code is based on the code open sourced from Kim et al. [2016]. The models are trained on a single GPU. In the experiments, the dimension of the

Table 3.1: Corpus Statistics

Dataset	#Token	Vocabulary Size
PTB	1M	10K
44M	44M	60K
WMT12	58M	35K
ACLW-Spanish	56M	152K
ACLW-French	57M	137K
ACLW-English	20M	60K
ACLW-Czech	17M	206K
ACLW-German	51M	339K
ACLW-Russian	25M	497K
BillionW	799M	793K

embeddings is the same as the number of hidden states in the LSTM model. Perplexity (PPL) is used to evaluate the model performance. Perplexity over the test set with length of T is given by

$$\text{PPL} = \exp(-\frac{1}{T}\sum_{i=1}^{T}\log(p(w_i|w_1,\cdots,w_{i-1})).$$

When counting the number of parameters, for convenience, we did not include the mapping table that maps each word to its sub-vector ids. In all the experiments, the mapping table is fixed before the training process. For particularly large values of K, the mapping table's size could be larger than the size of the parameters in its embedding layer. It is possible to replace the mapping table with hash functions that are done in HashNet [Chen et al., 2015]. We added end of sentence tokens to all the datasets with the exception of the experiments in Table 3.4. Those experiments omit the end of sentence token for comparison with other baselines.

Similar to the work in [Jozefowicz et al., 2016], compressing the output layers turned out to be more challenging. We first reported the results when just compressing the input layer, and then reported the results when both the input layers and the output layers were compressed. In the end, we did the reranking experiments.

Table 3.2: Validation and test perplexities on PTB with 300 hidden nodes, K=10, K is the number of sub vectors each word has, and M is the total number of unique sub vectors.

Model	**Dropout**	**Test**	**Size**
NE	0	89.54	1
NE	0.1	88.56	1
NE	0.2	88.33	1
NE	0.5	91.10	1
SE (M=20K)	0	89.34	20%
SE (M=20K)	0.1	**88.19**	20%
SE (M=10K)	0	89.06	10%
SE (M=10K)	0.1	88.37	10%
SE (M=6.25K)	0	89.00	6.25%
SE (M=5K)	0	89.54	5%

Table 3.3: Validation and test perplexities on PTB with 650 hidden nodes, K=10, K is the number of sub vectors each word has, and M is the total number of unique sub vectors.

Model	**Dropout**	**Test**	**Size**
NE	0	85.33	1
NE	0.1	82.59	1
NE	0.2	83.51	1
NE	0.5	82.91	1
SE (M=1K)	0	82.62	1%
SE (M=5K)	0	82.41	5%
SE (M=5K)	0.1	**81.14**	5%
SE (M=10K)	0	82.14	10%

3.3.1 Experiments on Slim Embedding for Input Layer

For the input layer, we compared two cases. The first case used the original word embedding (NE), and the second case compressed the input embedding layer with different ratio (SE). The first case was very strong baseline, used the same number of hidden states, used the same full softmax layer and had many more parameters. We first reported the results on the Penn Treebank (PTB) dataset. For PTB, the vocabulary size is 10k, and has 1 million words.

Tables 3.2 and 3.3 show the experimental results on PTB corpus when using 350 and 650 hidden nodes respectively. In both tables, the column Dropout denotes the dropout probability that is used from the input embedding layer to the hidden layer. Size is the number of parameters in the compressed input word embedding layer relative to the original input word embedding. The experiment on the input layer shows the compression of the input layer has almost no influence on the performance of the model. The SE model with 650 hidden states managed to keep the PPL performance almost unchanged even when the input layer only used 1% of trainable parameters. When the input layer was trained using dropout, it gave better results than the baseline, while using less parameters.

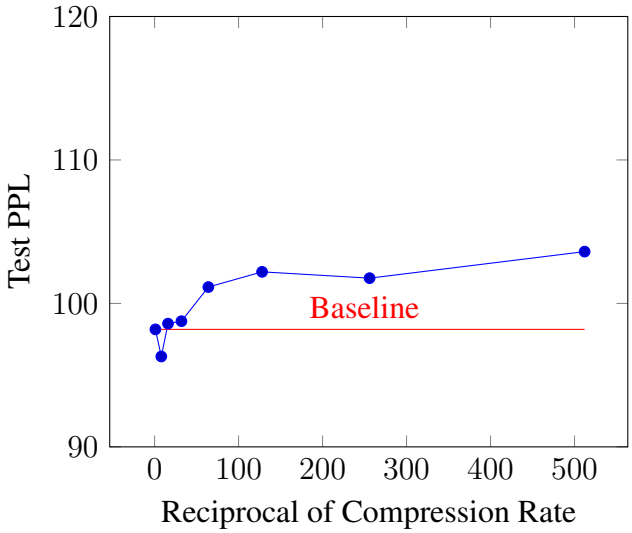

Figure 3.2: Test perplexities on 44M with 512 hidden nodes and each 512-dimensional input embedding vector was divided into eight parts. Only input word embedding layer was compressed.

Fig 3.2 and Fig 3.3 show the results on 44M Giga world sub-corpus where 512 hidden notes were used in the two-layer LSTM model. Baseline denotes the result using the original LSTM model. Fig 3.2 shows the perplexity results on both validation and test data sets, where we divided each word input embedding vector into eight sub-vectors ($K = 8$), and varied the number of new embedding sub-vectors, M, thus varying the compressed model size, i.e., compression ratio, from 1 to $1/512$.

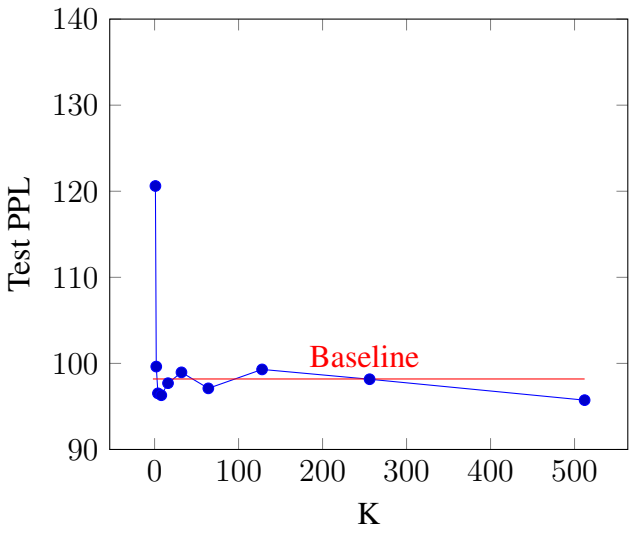

Figure 3.3: Test perplexities on 44M with 512 hidden nodes with 1/8 original Size. Only input word embedding layer is compressed.

The perplexity results remained almost the same and were quite robust and insensitive to the compression ratio: they decreased slightly to a minimum of 96.30 when the compression ratio changed from 1 to 1/8, but increased slightly to 103.61 when the compression ratio reached $1/512$. Fig 3.3 shows the perplexity results on both validation and test data sets, where we divided each word input embedding vector into different numbers of sub-vectors from 1 to 512, and at the same time varied the size of new embedding vectors, M, so as to keep the compression ratio constant, $1/8$ in this case. The perplexity results remained almost the same, and reached the minimum when $k = 8$, and were quite robust and insensitive to the size of the sub-vector except in the case where each word contained only one sub-vector but we still compressed the model into a 1/8 size of original matrix, i.e. $k = 1$. In this case, multiple words shared identical input embeddings, which led to worse perplexity results as we expected.

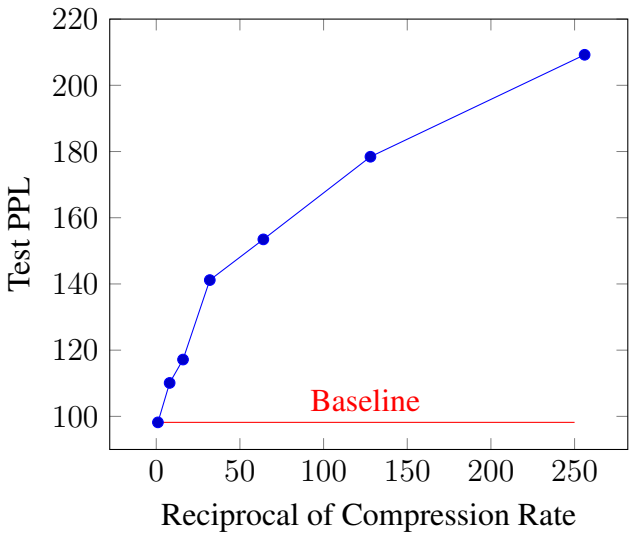

Figure 3.4: Test perplexities on 44M with 512 hidden nodes and each 512-dimensional vector divided into eight parts. Both input and output embedding layers were compressed.

3.3.2 Experiments on Slim Embedding for Both Input and Output Layers

In this section we report experimental results when both input and output layers are compressed using our proposed approach.

Fig 3.4 and Fig 3.5 show the results on the 44M corpus where 512 hidden nodes are used in the two layers of the LSTM model. Baseline denotes the result using the original LSTM model. Similarly Fig 3.4 shows the perplexity results on both validation and test data sets, where we divided each word input embedding vector into eight sub-vectors ($K = 8$), and varied the size of new embedding vectors, M, thus varying the compressed model size, i.e., compression ratio, from 1 to $1/256$.

Unlike the case when only the input embedding layer was compressed, the perplexity results become monotonically worse when the compression ratio was changed from 1 to $1/512$. Similarly to the case of only compressing the input embedding layer, Figure 3.5 shows the perplexity results on both validation and test data sets, where we divided each word input embedding vector into different sub-vectors from 1 to 512, and at the same time

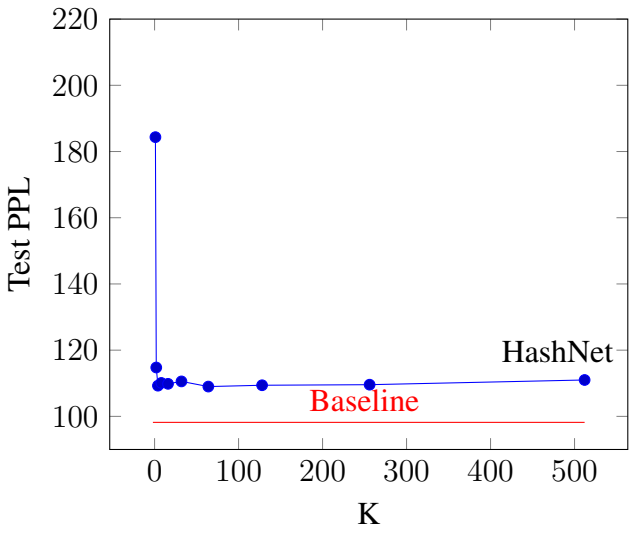

Figure 3.5: Test perplexities on 44M with 512 hidden nodes when embedding compressed to 1/8. The whole model size is less than 20% of the baseline.

varying the size of new embedding vectors, i.e., varying M, thus keeping compressing ratio to be constant, $1/8$ in this case. The perplexity results almost remained the same, and reached to a minimum when $k = 4$, and were quite robust and insensitive to the size of the sub-vector except in the case where each word contained only one sub-vector, but we still compressed the model into a 1/8 size of original matrix, i.e. $k = 1$. In this case, multiple words shared identical input embeddings, which led to worse perplexity results, as we expected.

Good perplexity results on PTB corpus were reported when parameter tying was used at both input and output embedding layers [Inan et al., 2016, Press and Wolf, 2017, Zilly et al., 2017, Zoph and Le, 2017]. However, we didn't observe further perplexity improvement when both parameter sharing and tying were used at both input and output embedding layers.

We next compared our model with LightRNN [Li et al., 2016], which also focuses on training very compact language models and reports the best result we got on the one-billion-word dataset. SE denotes the results using compressed input and output embedding layers. Table 3.4 shows the results of our model. Because these datasets have very different

vocabulary sizes, we used different compression rates for the models in order to make the model smaller than LightRNN, yet still have better performance. In these experiments, we changed to NCE training and tuned the parameters with the Adagrad [Duchi et al., 2011] algorithm. NCE helps reduce the memory usage during the training process and also speeds up the training process. In the one-billion-word experiments, the total memory used on the GPU was about 7Gb, and was smaller if a larger compression rate was used. We used a fixed, smoothed unigram distribution (unigram distribution raised to 0.75) as the noise distribution. Table 3.5 shows our results on the one-billion-word dataset. For the two-layer model, the compression rate for the input layer was 1/32 and the output layer was 1/8, and the total number of parameters was 322 million. For the three-layer model, the compression rates for the input and the output layer were 1/32 and 1/16, and the total number of parameters was 254 million. Both experiments took about seven days of training on a GTX 1080 GPU. Jozefowicz et al. [2016] suggests importance sampling could perform better than the NCE model, for the IS experiment we used 4000 noise samples for each mini-batch, and the PPL decreased to 38.3 after training for 8 days. As far as we know, the 3-layer model is the most compact RNN language model that has a perplexity below 40 on this dataset.

3.3.3 Machine Translation Reranking Experiment

We wanted to see whether the compressed language model would affect the performance of machine translation reranking. In this experiment, we used the Moses toolkit [Koehn et al., 2007] to generate a 200-best list of candidate translations. Moses was configured to use the default features, with a 5-gram language model. Both the language and translation models were trained using the WMT12 data [Callison-Burch et al., 2012], with the Europarl v7 corpus for training and newstest2010 for validation, both lowercased. The scores used

Method	English/#P	Russian/#P	Spanish/#P	French/#P	Czech/#P	German/#P
HSM [Kim et al., 2016]	236/25M	353/200M	186/61M	202/56M	701/83M	347/137M
C-HSM [Kim et al., 2016]	216/20M	313/152M	169/48M	190/44M	578/64M	305/104M
LightRNN [Li et al., 2016]	191/17M	288/19M	157/18M	176/17M	558/18M	281/18M
SE	187/7M	274/19M	149/8M	162/12M	528/17M	261/17M

Table 3.4: PPL results in test set for various linguistic datasets on ACLW datasets. Note that all the SE models only use 300 hidden states.

Model	Perplexity	#P[Billions]
Interpolated Kneser-Ney 5-gram [Chelba et al., 2014]	67.6	1.76
4-layer IRNN-512 [Le et al., 2015]	69.4	
RNN-2048 + BlackOut sampling [Ji et al., 2016]	**68.3**	
Sparse Non-negative Matrix Language Model [Pelemans et al., 2016]	52.9	33
RNN-1024 + MaxEnt 9-gram [Chelba et al., 2014]	51.3	20
LSTM-2048-512 [Joulin et al., 2017]	43.7	0.83
LightRNN [Li et al., 2016]	**66.0**	0.041
2-layer LSTM-8192-1024 [Jozefowicz et al., 2016]	30.6	1.8
2-layer LSTM-8192-1024 + CNN inputs [Jozefowicz et al., 2016]	30.0	1.04
2-layer LSTM-8192-1024 + CNN inputs + CNN softmax [Jozefowicz et al., 2016]	39.8	0.29
LSTM-2048 Adaptive Softmax [Joulin et al., 2017]	**43.9**	>0.29
2-layer LSTM-2048 Adaptive Softmax [Joulin et al., 2017]	**39.8**	
GCNN-13 [Dauphin et al., 2017]	**38.1**	
MOE [Shazeer et al., 2017]	28.0	>4.37
SE (2-layer 2048 LSTM NCE)	**39.9**	0.32
SE (3-layer 2048 LSTM NCE)	**39.5**	0.25
SE (3-layer 2048 LSTM IS)	**38.3**	0.25

Table 3.5: Perplexity results for single models on BillionW. Bold numbers denote using single GPU.

Table 3.6: Reranking Experiment

	Baseline	NE	SE
PPL	251.7	124.1	134.8
BLEU	25.69	26.11	26.25

for reranking were linear combinations of the Moses features and the language models. ZMERT [Zaidan, 2009] was used to determine the coefficients for the features.

We trained a two-layer LSTM language model with 512 hidden states, and a compressed language model that compressed the input layer to 1/8 and the output layer to 1/4 using NCE. For the baseline, we reranked the n-best list using only the Moses feature scores that include 5-gram having a perplexity of 251.7 on test data, yielding a BLEU score of 25.69. When we added the normal LSTM language model having a perplexity of 124 on test data as another feature, the BLEU score changed to 26.11, and for the compressed language model having a perplexity 134 on test data, the BLEU score changed to 26.25, which has a nearly indistinguishable change in performance from the normal LSTM language model.

3.3.4 Closest Word

Slim embedding layer introduces random parameter sharing into the model. It would be interesting to see the nearest neighbor of the word embeddings.

Cosine similarity is used as the distance metric. Given two word vectors v_i, v_j, the distance is computed as

$$-\frac{\sum_k v_{ik} * v_{jk}}{\sqrt{\sum_k v_{ik}^2}\sqrt{\sum_k v_{jk}^2}}$$

The method to find the closest word for a word is to compute all the distance between other words and then find the words with the shortest distance.

Some of the examples are listed in Table 3.7. As expected, the results still make

Table 3.7: Nearest neighbor of word embeddings

Word	Top 1	Top 2	Top 3	Top 4
While	while	Although	whilst	Though
his	her	their	its	the
you	we	You	they	I
Richard	Robert	John	David	Edward
Bob	Mike	Ed	Alan	Jonathan
Emily	Megan	Rachael	Baker	Keith
hate	love	hates	hated	loves
idea	notion	concept	possibility	proposal
word	phrase	words	term	wish
trading	trade	traded	trades	Trading
life	lives	career	death	Life
love	loves	hate	loved	passion
meaning	means	mean	saying	thought
top	Top	senior	prominent	supreme
wonderful	fantastic	lovely	great	excellent
Google	Apple	Microsoft	Facebook	Yahoo
breaking	broke	break	broken	smashing
time	day	moment	year	years
drawn	draw	drew	draws	attracted
happy	unhappy	pleased	glad	satisfied
size	length	age	shape	sizes
energy	electricity	power	oil	fuel

sense. Although quite different words could share sub-vectors, the similarity between similar words are still preserved.

3.3.5 Computational Efficiency

In this section, we compared the computational efficiency between the HashNet and SE models. We compare the time spent on the output layer for each minibatch on Google's BillionW corpus during inference. Each minibatch contains 20 words and the number of hidden nodes in LSTM layer is 2048.

We reported the time used on both CPU and GPU. Table 3.8 shows the inference time usage. All the computations used 32-bit floating point numbers. On CPU, HashNet is

Table 3.8: Time Usage Comparison

Model	CPU(seconds)	GPU (milliseconds)
Uncompressed Model	2.7	38
HashNet	80.6	-
SE	0.7	25

slower than the normal uncompressed model, mainly due to: 1) the uncompressed model uses optimized matrix multiplication subroutines, and 2) the hash function used in HashNet is cheap, but it still has overhead compared with the uncompressed model. The SE model runs faster mainly because it uses matrix multiplication subroutines and has lower time complexity with the help of dynamic programming.

On GPU, SE's time usage is smaller than the uncompressed model when K is small. SE's inference has two steps, the first step is K matrix multiplications, and the second step is adding up the partial dot products. In the benchmark, the implementation uses Torch. A more optimized implementation is possible.

HashNet's focus is mainly on reducing the space complexity. If we wanted to make it faster, we could just cache the full matrix from HashNet, whose speed is the same as the uncompressed model. There are many techniques that could be used to make the inference faster, such as using low-precision floating point calculations. Because the model stays in its compressed form, the memory usage of SE during inference is much lower than the baseline.

Clustering Based Assignment of Sub-vectors

The word embedding vector is the distributed representation of the word. Experiments have shown that word vectors have additive compositionality [Mikolov et al., 2013]. The proposed random parameter sharing seems counter-intuitive. A more reasonable approach is that we first get all the words' embedding, which we dive into sub-vectors, and then do a clustering analysis. Then we could let the sub-vectors in the same cluster share the same sub-vector.

Then, the problem is whether we can get the word embedding before we train the language model. The obvious way to achieve this is simply first train the full language model, and then the embedding can be used to do the clustering analysis. The major drawback of this is that the space complexity of training of such a model will be the same as training the uncompressed language model.

It is very lucky that there are many ways to get good quality word vectors without training of the full language model [Mikolov et al., 2013]. The most popular method is to use word2vec. Although it has been shown that word2vec's loss function is not related to the perplexity of the model, they are still good tools for representation learning [Dyer, 2014]. There are several efficient implementations of this model. The best toolkit is FastText [Joulin et al., 2016], which also uses the Hogwild [Recht et al., 2011] algorithm to do

lock free parallel training of the model. The implementation is so efficient that the training could be done within minutes even when using large corpus. This enables training the word representations on a very much larger corpus. And it is easy to download pretrained word embeddings online.

4.1 Related Work

There has been many works that uses clustering algorithm to reduce the number of parameters in neural network models. Han et al. [2016] uses clustering algorithm to enable parameter sharing as one step of their compression algorithm. Joulin et al. [2016] uses product quantization to significantly reduce the memory usage of word embeddings. Both models require training of the full model and then do the clustering analysis. Our approach is different that we do clustering on the already trained word embeddings and then get the clustering assignments. We will train the model based on the clustering assignments, and the parameters in the embedding layers will still be initilized with random values in the beginning and will be updated during the training process.

4.2 Clustering of Sub-vectors

When using randomly sharing for the input layer, in the previous chapter, we have shown that the performance of the model is barely effected. Therefore, it is not reasonable to try clustering-based methods on the input layer. However, it is still interesting to see if we can get improved performance using clustering algorithm to do the sub-vector assignments.

There are two ways to get word embeddings, 1) Generate word embedding from the current training dataset, 2) Just download the pre-trained word embeddings trained from a much larger corpus. Previous work has shown that using pre-trained word embeddings on large corpus help improve the performance of neural language model. And for small

dataset like PTB, the dataset could be too small to achieve good quality word embeddings.

After we get the word embeddings, we split the each word embeddings into K different part. And then run K-means clustering algorithm on the sub-vectors. The sub-vectors that are in the same cluster share the same sub-vector.

4.3 Experiment Results

Table 4.1 shows the input layer result when using clustering based assignment. We can see that both datasets show improvements.

Table 4.1: Clustering sub-vector assignment on the input layer

Dataset	Baseline	Clustering
PTB	88.93	86.23
44M	98.85	94.51

On PTB dataset, the skipgram embedding trained just on PTB dataset dones't show improved perplexity. After switching to the 300D pre-trained common crawl word embedding downloaded from FastText website [Mikolov et al., 2018], the PPL is decreased. On 44M dataset, simply using the word embedding trained on 44M dataset can improve the PPL.

Table 4.2: Clustering sub-vector assignment on the output layer

Dataset	Baseline	Clustering
PTB	103.98	99.80
44M	114.39	112.84

Table 4.2 shows the result of using clustering based assignment on the output layer. On PTB dataset, the word embedding used to do clustering analysis also comes from Mikolov et al. 2018.

Table 4.3: Clustering sub-vector assignment on both input and output layers

Dataset	Baseline	Clustering
PTB	104.76	98.63
44M	108.89	104.97

Table 4.3 shows the result of using clustering based assignment on both input and output layers. For the PTB dataset, the word embedding used to do clustering analysis also comes from Mikolov et al. [2018].

4.4 Discussion

When compared with slim embedding layer with random parameter sharing, experimental results show that clustering based assignment of sub-vectors on the input and output layers of neural language model does help to improve model's performance. This result is expected from our understanding, where the quality of word embedding is important. For the PTB data set, many words in the vocabulary are not found in the common craw 2M word vocabulary. So for the words missing from the 2M words vocabulary, I just used random assignments, but it still helps to improve the PPLs.

Conclusion and Future Work

We came up with a space-efficient, structured parameter sharing method to compress word embedding layers. Experiments on several datasets showed good PPLs. The model is easy to implement and tune. On the output layer, we also reduced the time complexity of inference algorithm.

Embedding layers have been used in many tasks of natural language processing, such as sequence to sequence models for neural machine translation and dialog systems. It would be useful to explore the results of using this technique in these models.

There are many output layer optimization techniques, such as hierarchical softmax. It will be interesting to see the model performance comparison when using random parameter sharing.

A theoretical investigation is needed to explain why compressing the input embedding layer, but not the output embedding layer, has a minor impact on deteriorating the performance of the neural language model.

Finally it would be interesting to explore generative adversarial network (GAN) [Goodfellow et al., 2014] for language modeling.

Bibliography

Yoshua Bengio and Jean-Sébastien Senécal. Adaptive importance sampling to accelerate training of a neural probabilistic language model. *IEEE Transactions on Neural Networks*, 19(4):713–722, 2008.

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

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

Chris Callison-Burch, Philipp Koehn, Christof Monz, Matt Post, Radu Soricut, and Lucia Specia, editors. *WMT '12: Proceedings of the Seventh Workshop on Statistical Machine Translation*, Stroudsburg, PA, USA, 2012. Association for Computational Linguistics. http://www.statmt.org/wmt12/translation-task.html.

Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. One billion word benchmark for measuring progress in statistical language modeling. *INTERSPEECH*, pages 2635–2639, 2014.

Stanley F Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. *Computer Speech and Language*, 13(4):359–394, 1999.

Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *The 32nd International Conference on Machine Learning (ICML)*, pages 2285–2294, 2015.

Yunchuan Chen, Lili Mou, Yan Xu, Ge Li, and Zhi Jin. Compressing neural language models by sparse word representations. *The 54th Annual Meeting of the Association for Computational Linguistics, (ACL)*, pages 226–235, 2016.

R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *International Conference on Machine Learning*, pages 933–941, 2017.

Misha Denil, Babak Shakibi, Laurent Dinh, Nando de Freitas, et al. Predicting parameters in deep learning. In *Advances in Neural Information Processing Systems*, pages 2148–2156, 2013.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul): 2121–2159, 2011.

Chris Dyer. Notes on noise contrastive estimation and negative sampling. *arXiv preprint arXiv:1410.8251*, 2014.

Ronald Aylmer Fisher, Frank Yates, et al. Statistical tables for biological, agricultural and medical research. *Statistical Tables for Biological, Agricultural and Medical Research.*, (5th rev. ed), 1957.

Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in

recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 1019–1027, 2016.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, pages 2672–2680, 2014.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Joshua Goodman. Classes for fast maximum entropy training. In *IEEE International Conference on Acoustics, Speech, and Signal Processing, (ICASSP)*, volume 1, pages 561–564. IEEE, 2001.

Google. Computer, respond to this email., 2015. URL https://research.googleblog.com/2015/11/computer-respond-to-this-email.html.

Google. Chat smarter with allo, 2016. URL https://ai.googleblog.com/2016/05/chat-smarter-with-allo.html.

David Graff and Christopher Cieri. English gigaword ldc2003t05. *Linguistic Data Consortium, Philadelphia*, 2003.

Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

Michael Gutmann and Aapo Hyvärinen. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 1, page 6, 2010.

Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural

networks with pruning, trained quantization and huffman coding. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

Hakan Inan, Khashayar Khosravi, and Richard Socher. Tying word vectors and word classifiers: A loss framework for language modeling. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

Shihao Ji, SVN Vishwanathan, Nadathur Satish, Michael J Anderson, and Pradeep Dubey. Blackout: Speeding up recurrent neural network language models with very large vocabularies. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

Armand Joulin, Edouard Grave, Piotr Bojanowski, Matthijs Douze, Hérve Jégou, and Tomas Mikolov. Fasttext.zip: Compressing text classification models. *arXiv preprint arXiv:1612.03651*, 2016.

Armand Joulin, Moustapha Cissé, David Grangier, Hervé Jégou, et al. Efficient softmax approximation for gpus. In *International Conference on Machine Learning*, pages 1302–1310, 2017.

Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.

Yoon Kim, Yacine Jernite, David Sontag, and Alexander M Rush. Character-aware neural language models. *The 30th AAAI Conference on Artificial Intelligence (AAAI)*, 2016.

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, et al. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th annual meeting of the ACL on interactive poster and demonstration sessions*, pages 177–180. Association for Computational Linguistics, 2007.

Quoc V Le, Navdeep Jaitly, and Geoffrey E Hinton. A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941*, 2015.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Xiang Li, Tao Qin, Jian Yang, Xiaolin Hu, and Tieyan Liu. Lightrnn: Memory and computation-efficient recurrent neural networks. In *Advances In Neural Information Processing Systems*, pages 4385–4393, 2016.

Wang Ling, Chris Dyer, Alan W Black, Isabel Trancoso, Ramon Fermandez, Silvio Amir, Luis Marujo, and Tiago Luis. Finding function in form: Compositional character models for open vocabulary word representation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1520–1530, 2015.

Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2): 313–330, 1993.

Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Interspeech*, volume 2, page 3, 2010.

Tomáš Mikolov, Stefan Kombrink, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Extensions of recurrent neural network language model. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5528–5531. IEEE, 2011a.

Tomas Mikolov, Stefan Kombrink, Anoop Deoras, Lukar Burget, and Jan Cernocky. Rnnlm-recurrent neural network language modeling toolkit. In *The 2011 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, pages 196–201, 2011b.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*, pages 3111–3119, 2013.

Tomas Mikolov, Edouard Grave, Piotr Bojanowski, Christian Puhrsch, and Armand Joulin. Advances in pre-training distributed word representations. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*, 2018.

Yasumasa Miyamoto and Kyunghyun Cho. Gated word-character recurrent language model. *The 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1992–1997, 2016.

Andriy Mnih and Yee W Teh. A fast and simple algorithm for training neural probabilistic language models. In *The 29th International Conference on Machine Learning (ICML)*, pages 1751–1758, 2012.

Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In *Tenth International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 5, pages 246–252. Citeseer, 2005.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics (ACL)*, pages 311–318. Association for Computational Linguistics, 2002.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013.

Joris Pelemans, Noam Shazeer, and Ciprian Chelba. Sparse non-negative matrix language modeling. *Transactions of the Association for Computational Linguistics*,

4:329–342, 2016. URL https://transacl.org/ojs/index.php/tacl/article/view/561.

Ofir Press and Lior Wolf. Using the output embedding to improve language models. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, pages 157–163, Valencia, Spain, April 2017. Association for Computational Linguistics.

Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

Ronald Rosenfeld. Two decades of statistical language modeling: Where do we go from here? *Proceedings of the IEEE*, 88(8):1270–1278, 2000.

Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016. https://arxiv.org/abs/1609.04747.

Abigail See, Minh-Thang Luong, and Christopher D Manning. Compression of neural machine translation models via pruning. *arXiv preprint arXiv:1606.09274*, 2016.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017. URL https://openreview.net/pdf?id=B1ckMDqlg.

Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

Jun Suzuki and Masaaki Nagata. Learning compact neural word embeddings by parameter space sharing. In Subbarao Kambhampati, editor, *The Twenty-Fifth International Joint Conference on Artificial Intelligence, (IJCAI)*, pages 2046–2052. IJCAI/AAAI Press, 2016.

Ming Tan, Wenli Zhou, Lei Zheng, and Shaojun Wang. A scalable distributed syntactic, semantic, and lexical language model. *Computational Linguistics*, 38(3):631–671, 2012.

Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

Will Williams, Niranjani Prasad, David Mrva, Tom Ash, and Tony Robinson. Scaling recurrent neural network language models. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 5391–5395. IEEE, 2015.

Wired. Apple is bringing the AI revolution to your iPhone, 2016. URL https://www.wired.com/2016/06/apple-bringing-ai-revolution-iphone/.

Omar F. Zaidan. Z-MERT: A fully configurable open source tool for minimum error rate training of machine translation systems. *The Prague Bulletin of Mathematical Linguistics*, 91:79–88, 2009.

Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

Julian Georg Zilly, Rupesh Kumar Srivastava, Jan Koutnık, and Jürgen Schmidhuber. Recurrent highway networks. In *International Conference on Machine Learning*, pages 4189–4198, 2017.

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.

Barret Zoph, Ashish Vaswani, Jonathan May, and Kevin Knight. Simple, fast noise-contrastive estimation for large rnn vocabularies. In *The 15th Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. NAACL/HLT, 2016.