TensorFlow Applied to Neural Network Language Models

Zihan Zhang

Master of Science

Data Science

School of Informatics

University of Edinburgh

2018

Abstract

Machine learning frameworks help to efficiently implement machine learning algorithms. TensorFlow is one of the popular machine learning frameworks with static dataflow graphs. In this project, we explore the performance of TensorFlow in implementing neural network language models in terms of perplexity and speed. The standard long short-term memory (LSTM) recurrent neural networks (RNN) with non-recurrent dropout are implemented as the baseline. Then we apply mixture of Softmaxes (MoS) to increase the capacity of the models and improve the perplexities. The hyperparameter of MoS models are optimized by random search. The best MoS model has lower perplexity and higher speed than the baseline model. However, MoS slows down the convergence rate of the models and more training time has to be spent to achieve to lower perplexity. Hence, several tricks are applied to speed up training. The final model has higher speed and lower perplexity, compared to the baseline model. Finally, we implement the final MoS model in PyTorch and get a similar perplextiy compared with the TensorFlow model. However, the speed of the former is faster than the latter because of different ways to implement the LSTM cell.

Acknowledgements

Many thanks to my supervisors, Ben Allison and Tania Bakhos, for instructing me to finish this project and giving me many suggestions. I would also like to thank my groupmates, Xin Chen, Abhijit Singh and Cong Chen who explore other machine learning frameworks on neural network language models, for sharing ideas and experimental results with me. I would like to express thanks to my parents and friends who support me in the duration of carrying out this project.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Zihan Zhang*)

Table of Contents

Chapter 1

Introduction

Machine learning frameworks are the programming language packages that help to implement machine learning models by array-based computation and automatic gradient computation. A good machine learning framework could save training time and computational resources and enhance the accuracy of the models. Due to different design principle, different machine learning frameworks could perform very differently in building the same model in terms of effectiveness and efficiency. Hence, it is important for practitioners to choose an appropriate machine learning framework to achieve high efficiency and good performance, according to the specific models.

TensorFlow (Abadi et al., 2015) is one of the most famous and popular machine learning frameworks, which can be used to build most of machine learning models. TensorFlow constructs entire dataflow graphs of machine learning algorithms before feeding data and training parameters. In each iteration of supervised learning, data goes through the dataflow graphs to compute losses of algorithms, which is called forward propagations; then TensorFlow automatically computes gradients of all parameters that can be trained and updated them by some rules, which is called back propagations. TensorFlow can employ GPUs and parallelize computation graphs on GPU kernels to significantly increase speeds of training compared with CPU. Also, computation graphs can be parallelized on multiple GPUs, which further speed up training.

In this project, we investigate the efficiency and effectiveness of TensorFlow applied on neural network language models (NNLMs), especially with Mixture of Softmaxes (MoS) (Yang et al., 2018). Traditional language models are designed to compute the probability of a sequence of words conditioned on a number of previous words. Xu and Rudnicky (2000) proposed using artificial neural networks to learn language models, i.e. neural network language models (NNLMs). Then recurrent neural networks (RNN) were used in language models (Mikolov et al., 2010) to predict the next word or sentence given the context, taking into account the time series of the words. Yang et al. (2018) identified that the key limitation of a standard RNN language model is the softmax function (Bridle, 1990) before the output layer, and proposed MoS technique to address this problem. MoS is a novel technique which addresses the Softmax bottleneck and enhances expressiveness of RNN models. However, the only publically existing implementation of MoS is in PyTorch, a machine learning framework with dynamic computational graphs, and no implementations in TensorFlow can be found. Hence, it is a good architecture to test the frameworks on. We implement NNLMs with and without MoS to explore the effect of MoS and evaluate the usability of of TensorFlow. Then we apply several tricks to speed up model training in TensorFlow. Finally, we compared models implemented in TensorFlow with those implemented in PyTorch.

This dissertation is arranged as follows. In Chapter 2 we introduce the background of neural language models and machine learning frameworks, focusing on TensorFlow. Then we state the objectives and the evaluation criteria in Chapter 3. Chapter 4 introduces data pre-processing and language modelling. In Chapter 5, we introduce the baseline models with different configurations. Then we applied MoS on the baseline models, which is discussed in Chapter 6. To speed up training, we optimize the model by different strategies in Chapter 7. The following chapter compares the performance of TensorFlow and PyTorch on language models. The last two chapters discuss the possible future works and make a conclusion, respectively.

Note: Part of Chapter 1, 2, 6.1 and 6.2 are modified from my IPP report.

Chapter 2

Background

The related work and development history for neural language language models and TensorFlow are introduced in this chapter. Chapter 2.1 compares the features of symbolic and imperative machine learning frameworks, and describes the design philosophy of TensorFlow, which is the framework we focus on. Chapter 2.2 introduced the characteristics of NNLMs and the optimization approaches.

Note: This chapter is based on my IPP report.

2.1 Machine Learning Frameworks

Machine learning frameworks can be classified into two classes - symbolic and imperative frameworks. Symbolic frameworks deploy static computational graphs to implement machine learning algorithms. The code can be divided into two chunks. One is for building static graphs and the other is for inputting realistic data and running graphs. Imperative frameworks adopt a different strategy, where codes combine model building and running. Among popular machine learning frameworks, Theano (Al-Rfou et al., 2016), TensorFlow, Keras, Caffe (Jia et al., 2014) and MXNet (Chen et al., 2015) with symbolic API are symbolic frameworks, while Torch (Collobert et al., 2002), Pytorch (Paszke et al., 2017), DyNet (Neubig et al., 2017), Chainer (Tokui et al., 2015) and MXNet with Gluon API are imperative frameworks.

For symbolic frameworks, a training process starts after an entire graph is built, which helps to globally programme the allocation of memory and the order of computations. In addition, this strategy makes static graphs reusable. However, for different inputs, the operations applied on them might be expected to be different. In this case, it is hard for symbolic frameworks to handle it. Actually, static graphs need to control dataflows by specific condition/loop functions designed by frameworks, such as `tf.cond` and `tf.while_loop` in TensorFlow, other than the default condition/loop statements of programming languages, such as **`if`**, **`for`** and **`while`** in python. This makes codes complicated and hard to read. All situations need to be considered before computation actually happens, which is usually not a easy thing. By contrast, imperative frameworks could deal with this problem easily. Dynamic graphs are built at the beginning of every iteration of training, so it is possible to apply different operations on different inputs. The default condition and loop statements can be used to control dataflows in imperative frameworks, which makes codes neat. However, building graphs repeatedly could be more costly.

Static graphs of symbolic frameworks only allows inputs of the same shape. However, in many cases of modelling, we need inputs to have different shapes. It is hard for these symbolic frameworks to deal with these models efficiently. To deal with this problem, symbolic frameworks reshape the inputs by bucketing and padding, and input them into computation graph as a batch. Dynamic graphs of imperative frameworks deal with this problem better. They allows inputs from different batches to have different shapes. However, in each batch the shapes of inputs need to be the same. Hence, Looks et al. (2017) proposed dynamic batching technique to address this problem. Dynamic batching takes the different graphs of inputs and rewrites the graphs by batching together the operations of the same depth. This technique is implemented in TensorFlow as a library called TensorFlow Fold.

TensorFlow (Abadi et al., 2015) is a symbolic framework, which is developed from DistBelief (Dean et al., 2012). DistBelief is large-scale distributed deep networks that use directed acyclic graphs (DAGs) to build models, where workers, stateless nodes, are deployed for computation, and parameter servers, stateful nodes, are deployed to store the model parameters that can be updated in a training process. However, although DistBelief is enough for users to build simple models, it is hard to be used to implement many advanced algorithms (Abadi et al., 2016). DistBelief layers are classes of C++, so it is not friendly to Python users to define new layers. In addition, some optimization rules other than stochastic gradient descent (SGD) and some machine learning algorithms including recurrent neural networks (RNN) are hard to be defined in DistBelief. Therefore, a more flexible machine learning framework, TensorFlow, came into being.

TensorFlow uses a static graph to represent dataflow in the model, where the edges are tensors and vertices are operations. Tensors are actually the high-dimensional arrays, which play the roles of inputs or outputs of some operations. Inputs are defined by `tf.placeholder` in the graphs. The operation vertices could do computation on tensors, and also output results in the form of tensors. Variables are a special kind of operation, which are used to contain the trainable parameters in the model. Variables, which is defined by `tf.Variable`, do not have the inputs but could update the stored parameters within training phases. A dataflow graph would only be run if a session is built and the input tensors are fed by actual data. Although static graphs have limitations, TensorFlow is very flexible and allows users to build advanced models easily. The operations in TensorFlow are built in C++, making computations efficient. TensorFlow provides a Python API, which makes it easy to implement machine learning algorithms. TensorFlow could deploy many machines in a cluster or many multicore CPUs or GPUs in one machine to parallelize dataflow and accelerate computation. Furthermore, it could be used in a wide range of platforms, from large distributed clusters to mobile devices.

Although there are many frameworks available, there are few comparison studies on them. In fact, different frameworks may be good at different tasks. It may be hard for practitioners to choose a proper framework on specific tasks. Therefore, we explore the performance of TensorFlow on neural network language models (NNLMs) compared with PyTorch in this project.

2.2 Neural Network Language Models

The main modelling achitecture we implement is a language model with Mixture of Softmax (MoS) proposed by Yang et al. (2018).

Xu and Rudnicky (2000) were the first to apply artificial neural networks on language models. In the networks, input words are encoded to a big vocabulary matrix, in which each word is a vector with the same length. The loss function of language models is the logarithm of perplexity (see Chapter 3), and the activation function is Softmax function (Bridle, 1990) that computes the probabilities of the next word. The paper only experimented with a single-layer network, and trained it by back propagation. Due to the high dimension of the vocabulary, the amount of the parameters is very large, which will lead to a high risk of overfitting. This problem was addressed by Bengio et al. (2003), who proposed word embeddings. The first layer of the NNLM uses a shared mapping matrix to map input words to word feature vectors. Then the feature vectors are joined to be the inputs of the next hidden layer. The outputs of the hidden layer and the original word features are input into the output layer, where Softmax function makes sure the output probabilities sum to 1. The mapping matrix and the parameters in the networks are trained together.

Mikolov et al. (2010) applied simple RNN on language models. Word sequences are inputs of RNN, so time series in contexts are taken into account in the networks. The words in the context are input in order into the hidden units at the time stamps. Simultaneously, the hidden units at the previous time stamps are also input to the hidden units at the next time stamps. The outputs of the hidden units will be input to the output layer. Long short-term memory (LSTM) (Hochreiter and Schmidhuber, 1997) was introduced to deal with the gradient vanishing problem caused by long continuous multiplications of the same weight matrix in back propagation (Bengio et al., 1994). LSTM uses input gate, output gate and forget gate to change continuous matrix multiplications to element-wise multiplications. Large LSTM tends to overfit, while dropout strategy (Hinton et al., 2012), which is commonly used in deep models to prevent overfitting, does not work well in LSTM, because dropout may reduce the ability of recurrent connections to model long sequences. Zaremba et al. (2014) addressed this problem by only applying dropout to non-recurrent connections, which is used as the baseline in this project. However, this strategy cannot regularize recurrent connections. Variational dropout (Gal and Ghahramani, 2015) keeps the dropout masks at each time stamp the same and regularizes all connections. (Press and Wolf, 2016) tied the embedding matrix with the weight matrix in the output layer, in order to reduce the amount of parameters and increase the generalization of the model. (Kuchaiev and Ginsburg, 2017) factorized the combined weight matrices in LSTM layers, which decreased the amount of parameters and sped up training while led to a little lower accuracy. Furthermore, Merity et al. (2017) applied DropConnect (Wan et al., 2013) on weights of recurrent connections. Instead of randomly setting a subset of activations to 0 by dropout, DropConnect randomly sets a subset of weights to be 0.

Yang et al. (2018) identified that the key limitation of a standard RNN language model is the Softmax function (Bridle, 1990) in the output layer, and proposed MoS technique to address this problem. MoS breaks the constraint of the rank of the last hidden layer, and makes the rank of the outputs arbitrarily large. This is the main strategy we apply to improve our models. In addition to simple MoS, we implemented weight tying (Press and Wolf, 2016) and weight matrix factorization (Kuchaiev and Ginsburg, 2017) to reduce the number of hyperparameters to speed up training and reduce the risk of overfitting.

Chapter 3

Objectives

The aim of this project is to verify the efficiency and effectiveness of TensorFlow on neural network language models. NNLMs are used to predict words given the corresponding previous word sequence. We implement NNLMs using TensorFlow 1.4.1 on GPUs, and optimize and evaluate the models in terms of accuracy and efficiency.

The evaluation criterion of model accuracy is perplexity (ppl.), i.e. the N^{th} root of the inverse of probability of a series of predicted words based on a specific corpus. The formula to calculate perplexities is

$$ppl(sentence) = \sqrt[N]{p[w_1, w_2, ..., w_N]^{-1}} \tag{3.1}$$

where N is the length of the sentence and $\{w_n\}|_{n=1}^{N}$ are the words in the sentence. Lower perplexity means more accurate predictions.

We evaluate model efficiency through speeds of training and predicting. The dataset is split into three sets, i.e. training set, validation set and test set. The training stage includes training parameters on the training set and evaluating the model on the validation set, while in the test stage, the model is evaluated on the test set. We will use the number of words processed per second (wps) to measure the average speed in each training iteration, and use the whole training time and test time to evaluate the efficiency of the whole process.

The other important evaluation criteria, including the number of parameters and the peak memory usage of GPUs, are also taken into account. The number of parameters directly influence the training time and the generalization of the model. The peak memory usage is proportional to the number of parameters and can show the potential of GPUs. We will also evaluate TensorFlow by the difficulties and limitations we faced in the building of NNLMs.

Chapter 4

Data Preprocessing and Modelling

This chapter introduces how to preprocess raw texts and model them by recurrent neural networks (RNN) (Elman, 1990), especially long short-term memory (LSTM) RNN (Hochreiter and Schmidhuber, 1997).

4.1 Data Preprocessing

Raw texts cannot be directly inputted into RNN, because RNN can only make computations on numbers. Hence, we should preprocess raw texts and convert them into numbers that can be dealt with by RNN. The logistic inputs of RNN should be sentences or word sequences, where words are inputted in order at each time stamp. The first thing we do is to tokenize the words in the texts. That is to say, each unique word is given a unique integer ID. However, integer IDs have linear relations which does not exist in raw texts. For example, if the IDs of ”apple”, ”pear”, ”orange” are 1, 2, 3, respectively, $1+2=3$ stands for ”apple + pear = orange”, which is not true in reality. To deal with this problem, we one-hot encode the IDs of the words, making each ID a vector. Each word vector only contains 0 at all positions except the position of its ID where the element is 1. The length of each word vector is the size of the vocabulary, i.e. the number of unique words in the raw text. The size of the vocabulary is usually large, so the networks need a large amount of parameters, which leads to a large amount of computations and memory. We convert the word vectors to much shorter vectors by multiplying a word embedding matrix. Finally these embedded words can be inputted into RNN. The whole process is shown in Figure 4.1.

The dataset we use in this project is Penn Treebank Dataset (Mikolov et al., 2010) that contains 2499 stories from the Wall Street Journal. The training set, validation

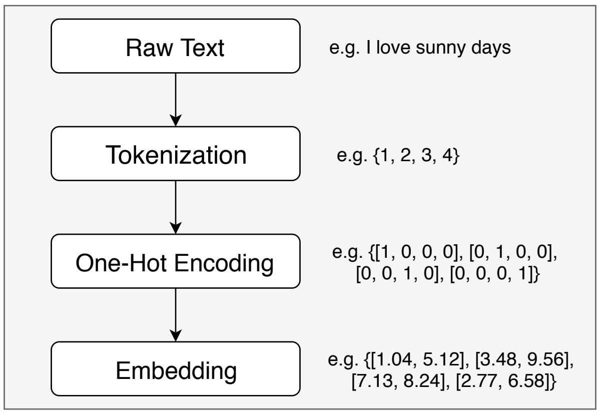

Raw Text
e.g. I love sunny days
Tokenization
e.g. {1, 2, 3, 4}
One-Hot Encoding
e.g. {[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]}
Embedding
e.g. {[1.04, 5.12], [3.48, 9.56], [7.13, 8.24], [2.77, 6.58]}

Figure 4.1: Raw texts preprocessing.

set and test set contains 887,521, 70,390 and 78,669 words, respectively, and the size of the vocabulary is set as 10,000. Since the dataset is too large, we divide it into many batches and input them iteratively into RNN. Each batch has a word sequence with the same length. However, the length of batches is too large for RNN, which will make computation of forward and backward propagations expensive. We divided each sequence into many mini sequences that are included in the same batch, shown in Figure 4.2. In each iteration, one sequence per batch is input into RNN, and the averaged gradients of all batches are used to update parameters. After several iterations, the whole dataset is traversed, which is called a training epoch. Usually we need many training epochs to update the parameters to achieve the optimum.

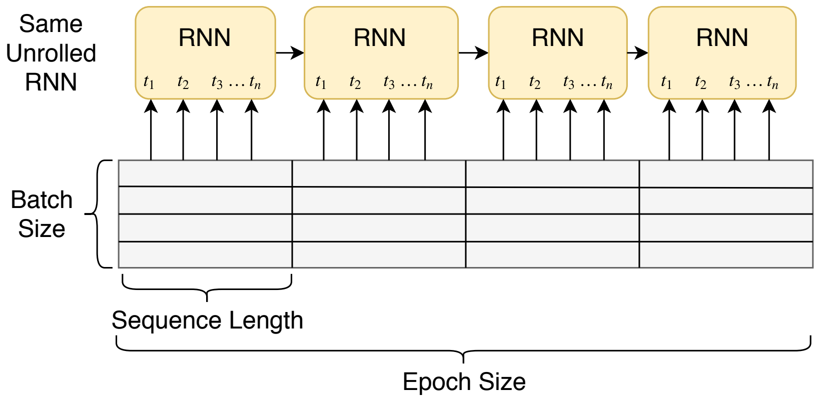


Figure 4.2: Divided data chunks.

4.2 Modelling Language by LSTM

To take the order of the word sequences into account, we use RNN to model the text, where each embedded word is input into a time stamp of RNN and the ground truth of the output is the next word. In the forward pass, we compute the predicted output using the current parameters. In the back propagation, the gradients, computed by the loss between predicted output and the ground truth, from all the time stamps are jointly used to update the parameters of RNN.

In the back propagation, the gradients of parameters are computed by the multiplication of the same recurrent weight matrix. For a long sequence, if the singular value of this matrix is larger than 1, the gradients explode; if the singular value is less than 1, the gradient vanish. We apply gradient clipping strategy (Pascanu et al., 2012) to deal with gradient explosion. If the square sum of the gradients S is larger than some threshold S_{max}, all gradients will be multiplied by $\frac{S_{max}}{S}$, in order to keep the square sum not too large. To deal with gradient vanishment, we use LSTM. LSTM converts inputs into four gates, which makes a hidden state and a cell state in each LSTM cell. The formulas of LSTM are as follows.

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \tag{4.1}$$

$$c_t = f \odot c_{t-1} + i \odot g, \quad h_t = o \odot \tanh(c_t) \tag{4.2}$$

where c_t and h_t are the hidden state and the cell state at time stamp t.

LSTM introduces a cell state in each cell, which makes the gradients flow along cell states instead of hidden weight matrices. The matrix multiplications are converted into element-wise multiplications by output gates (o in Formula 4.1 and Formula 4.2), which are different in all time stamps. This avoids continuous multiplications by the same matrix, thus solves the gradient vanishment problem.

Chapter 5

Baseline

The next two chapters describe the models we implement. This chapter describes the baseline, LSTM with non-recurrent dropout, and the hyperparameter configurations, followed by Chapter 6 which describes the Mixture of Softmax models. In this chapter, we discuss the difference among several LSTM implementations in TensorFlow. The various LSTM implementations that include optimization techniques will be compared to the baseline.

5.1 LSTM and Non-Recurrent Dropout

Modelling language by LSTM is easy to overfit, because no regularization strategy is applied on parameters and the model will be come very complicated with training. Dropout (Hinton et al., 2012) is a common regularization strategy in deep learning. In every iteration, we randomly set part of the hidden units to 0, and keep the other units. Dropout prevents the networks from relying too heavily on some neurons and thus reduces overfitting. Since we train different hidden units in different iterations, the final network could be considered as an ensemble network of many smaller networks, which makes the network more general. However, simple dropout cannot be used directly in RNN. because the recurrent units will be dropped at all time stamps, which makes it hard for RNN to learn long sequences. To address this problem, Zaremba et al. (2014) proposed to only apply dropout on all non-recurrent connections. That is to say, dropout is only applied between layers. For LSTM, Formula 4.1 becomes

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ Dropped(x_t) \end{pmatrix} \tag{5.1}$$

This strategy guarantees that however long the distance between the inputs and the outputs is, the number of dropout applied is the same. From Figure 5.1, in the same path between x_t and y_{t+n} the simple dropout drops n times at recurrent connections and twice at non-recurrent connections, while non-recurrent dropout only drops twice at non-recurrent connections.

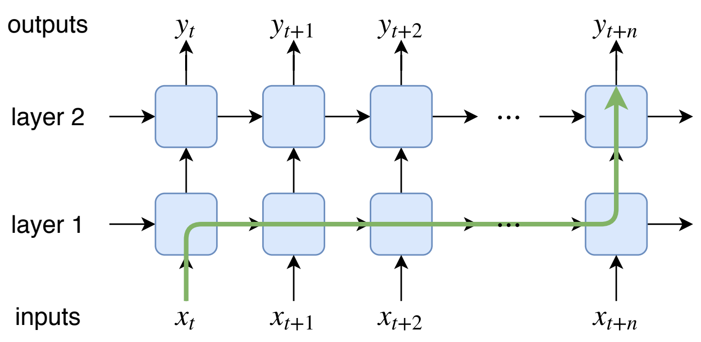


Figure 5.1: Dropout applied on one path.

Non-recurrent dropout regularizes well the parameters between layers and effectively reduces overfitting. However, the parameters in recurrent connections are not regularized, which is one of the limitations of this strategy.

5.2 TensorFlow RNN Implementations

We use LSTM with non-recurrent dropout as the baseline models, whose implementation is modified based on the TensorFlow RNN tutorial[1]. TensorFlow provides many implementations of LSTM cells. In this project, we explore the performance difference between the basic LSTM cell (`tf.contrib.rnn.BasicLSTMCell`) and the block LSTM cell (`tf.contrib.rnn.LSTMBlockCell`). The basic LSTM cell is implemented by pure TensorFlow statements, while the block LSTM cell is implemented by C++ and encapsulated by a TensorFlow operation. C++ operations are pre-compiled.

[1] `https://github.com/tensorflow/models/tree/master/tutorials/rnn/ptb`

Compared with Python operations, C++ operations save the time of Python interpretation. Hence, theoretically, the block LSTM should be more efficient than the basic LSTM cell. The LSTM cells are wrapped by dropout layers to implement non-recurrent dropout between layers.

5.3 Experiments and Results

We select three well-tuned configurations of hyperparameters, making a small, medium and large model using basic and block LSTM cells, respectively. The sizes of the models implies the number of parameters they contain. The larger models have more layers and hidden units, which increases the number of parameters. We apply Stochastic Gradient Descent (SGD) optimizer and learning rate schedules on all the models. That is to say, we keep the original learning rate in the first some training epochs and decrease the learning rate in every following epochs. The hyperparameter configurations for these models are in Appendix A.1. All the models are trained on one NVIDIA GTX1060 GPU, which has 6 GB RAM in total. The experimental results are shown in Table 5.1.

model	**small**		**medium**		**large**	
	basic	block	basic	block	basic	block
training ppl.	41.249	40.064	45.476	45.615	37.782	38.671
validation ppl.	120.166	118.463	87.127	87.579	82.486	83.306
test ppl.	114.396	113.304	83.179	83.648	78.195	79.241
training speed (wps)	16190	17770	8850	8700	2860	2780
training time (h)	0.239	0.208	1.176	1.197	5.050	5.209
test time (h)	0.076	0.065	0.069	0.065	0.083	0.086
total time (h)	0.315	0.273	1.245	1.262	5.133	5.295
#parameters (M)	4.652		19.775		66.022	
peak GPU RAM usage (GB)	0.065	0.065	0.940	0.928	4.830	4.771

Table 5.1: Performance of baseline models with different LSTM implementations, where wps means words per second.

From Table 5.1, it is clear that the larger models have lower perplexities than the smaller models, although they have much more parameters. The small models can be trained extremely fast, but the perplexities are much higher than the medium and large models. Hence, if a user values efficiency over perplexity, the small models are a better option. The medium models have a good balance of accuracy and efficiency, because they achieve low perplexities in a reasonable time. The large models spend much longer time to achieve a slight progress in perplexity. If perplexity is extremely important, large models are good choices. Training speed is inversely proportional to the number of parameters, while peak GPU RAM usage is proportional to the number of parameters. The reason is a larger number of parameters lead to more computations and occupy more memory. The larger models not only have lower training speed, but also spend more epochs to be trained, so the total times are much longer than the smaller models. The test time is much less than training time because no iterations in the test stage, and is not significantly influenced by the model size.

We find that basic and block LSTM cells do not have markable difference in terms of perplexity and speed. Block LSTM cells perform slightly better in the small models, while basic LSTM cells perform slightly better in the medium and large models. The memory usages of the models with block LSTM cells use slightly less GPU memory than the models with basic LSTM cells. This implies that the encapsulated C++ cell allocates and uses memory more efficiently.

These LSTM models with non-recurrent dropout were originally implemented by Zaremba et al. (2014) using Torch (Collobert et al., 2002) on an NVIDIA K20 GPU. Torch is a dynamic machine learning framework in Lua, which was transferred to Python to be PyTorch (Paszke et al., 2017) afterwards. NVIDIA K20 is a little weaker than NVIDIA GTX1060. We borrow the results from those experiments, according to Zaremba et al. (2014), and compare them with our results for TensorFlow in Table 5.2.

model	**framework**	**val ppl.**	**test ppl.**	**total time** (h)
medium	TensorFlow	87.1	83.2	1.2
medium	Torch	86.2	82.7	around 12
large	TensorFlow	82.5	78.2	5.1
large	Torch	82.2	78.4	around 24

Table 5.2: Performance of the medium and large baseline models in TensorFlow and Torch, where TensorFlow uses the basic LSTM cell.

From Table 5.2, we can see that the medium and large models in two frameworks have similar perplexities, where the slight difference is from the different initialized weights. However, then medium model in TensorFlow only take around 10% time of the same model in Torch, and the large TensorFlow model takes around 21% time of the large Torch model. The difference between two kinds of GPUs is small. Hence, TensorFlow has a great advantage in efficiency compared with Torch. This also verifies that TensorFlow is an effective and extremely efficient framework to build NNLMs.

Chapter 6

Mixture of Softmaxes

This chapter introduces the problem brought by Softmax function in RNN models and an advanced solution to this problem, Mixture of Softmaxes (MoS) in Section 6.1 and 6.2, which are based on my IPP report. Then the experiments are carried out to compare MoS models with baseline models in Section 6.3.

6.1 Softmax Bottleneck

Softmax function (Bridle, 1990) is commonly used in all kinds of RNN language models. Assuming that $\mathbf{h}_c$ is a hidden state, i.e. a d-dimensional vector, obtained from a fixed context c through a network and $\{\mathbf{w}_{x_m}\}\,|_{m=1}^{M}$ denote d-dimensional word embeddings for x_m before an output layer, the Softmax function for a prediction could be formulated as follows, where θ represents an exact prediction function from a function family $\mathcal{U}$.

$$P_\theta(x_i|c) = \frac{\exp \mathbf{h}_c^T \mathbf{w}_{x_i}}{\sum_{m=1}^{M} \exp \mathbf{h}_c^T \mathbf{w}_{x_m}} \tag{6.1}$$

The object of language models is to find a θ from $\mathcal{U}$ such that P_θ could emulate the true word distribution P_*. However, Yang et al. (2018) identified that Softmax function makes the expressiveness of language models constrained by the dimensions of the outputs. Let $\mathbf{H}_\theta = (\mathbf{h}_{c_1}^T \quad \mathbf{h}_{c_2}^T \cdots \mathbf{h}_{c_N}^T)^T$ be a matrix whose rows are hidden states from different contexts and $\mathbf{W}_\theta = (\mathbf{w}_{x_1}^T \quad \mathbf{w}_{x_2}^T \cdots \mathbf{w}_{x_M}^T)^T$ be a matrix whose rows are word embeddings. Due to the "linearity" of the Softmax function, the object becomes to find a matrix $\mathbf{A}' = \mathbf{H}_\theta \mathbf{W}_\theta^T$ to approximate the true logarithm probability distribution matrix $\mathbf{A}$, where $\mathbf{A}_{nm} = \log P_*(x_n|c_m)$. However, all matrices $\mathbf{A}$ such that $Softmax(A) = P_*$ have similar ranks, where the difference is no more than 1. In fact, if we make a set $\mathcal{A} =$

$\{\mathbf{A}|Softmax(\mathbf{A}) = P_*\}$, any element in $\mathcal{A}$ can be transformed by any other element by adding an arbitrary number to each row, which is called the "linearity" of $\mathbf{A}$. If we want $\mathbf{A}'$ approximates $\mathbf{A}$, the rank of $\mathbf{A}'$ should be similar with the rank of $\mathbf{A}$. However, the rank of $\mathbf{A}'$ is no more than d, the dimensions of the last hidden layer, because $\mathbf{A}' = \mathbf{H}_\theta \mathbf{W}_\theta^T$. Assuming that the true word distribution $\mathbf{A}$ is of high dimensions, d is usually much lower than that of the true word distribution. Hence, the model constrained by the Softmax function can only learn a low-dimensional distribution, and thus do not have enough capacity to model languages. See the detailed demonstrations of the above inferences and the "linearity" of Softmax function in Yang et al. (2018).

6.2 Solution to the Softmax Bottleneck

The straightforward solution is to increase the dimensions of the hidden layers, making the rank of $\mathbf{A}'$ closer to $\mathbf{A}$. This can increase the expressiveness of the model, but also leads to a much larger number of parameters. Numerous parameters will significantly increase the risk of overfitting, and thus hurts the generalization of the model. Therefore, choosing the number of hidden units is a tradeoff between expressiveness and generalization. In addition, numerous parameters make it hard to tune hyperparameters of the networks.

To address this problem, Yang et al. (2018) proposed Mixture of Softmaxes (MoS), which is a weighted average of Softmaxes showed as follows.

$$P_\theta(x_i|c) = \sum_{k=1}^{K} \pi_{c,k} \frac{\exp \mathbf{h}_{c,k}^T \mathbf{w}_{x_i}}{\sum_{m=1}^{M} \exp \mathbf{h}_{c,k}^T \mathbf{w}_{x_m}} \tag{6.2}$$

where $\pi_{c,k}$ is the mixture weight such that $\sum_{k=1}^{K} \pi_{c,k} = 1$. Both $\pi_{c,k}$ and $\mathbf{h}_{c,k}$ comes from the low-level hidden state $\mathbf{g}_t$, i.e. $\pi_{c_t,k} = \frac{\exp \mathbf{w}_{\pi,k}^T \mathbf{g}_t}{\sum_{k'=1}^{K} \exp \mathbf{w}_{\pi,k'}^T \mathbf{g}_t}$ and $\mathbf{h}_{c_k,k} = \tanh(\mathbf{W}_{h,k}\mathbf{g}_t)$, where $\mathbf{w}_{\pi,\mathbf{k}}$ and $\mathbf{W}_{\mathbf{h},\mathbf{k}}$ are the parameters that can be learned during training. The structures of networks with Softmax and MoS functions are shown in Figure 6.1.

The logarithm probability distribution matrix $\mathbf{A}$ becomes $\hat{\mathbf{A}}$ in Equation 6.3.

$$\hat{\mathbf{A}} = \log \sum_{k=1}^{K} \Pi_k \exp(\mathbf{H}_{\theta,k} \mathbf{W}_\theta^T) \tag{6.3}$$

According to the formula above, MoS does not keep the "linearity" of Softmax. Therefore, the expressiveness of the models gets rid of the limitation of dimensions of hidden layers.

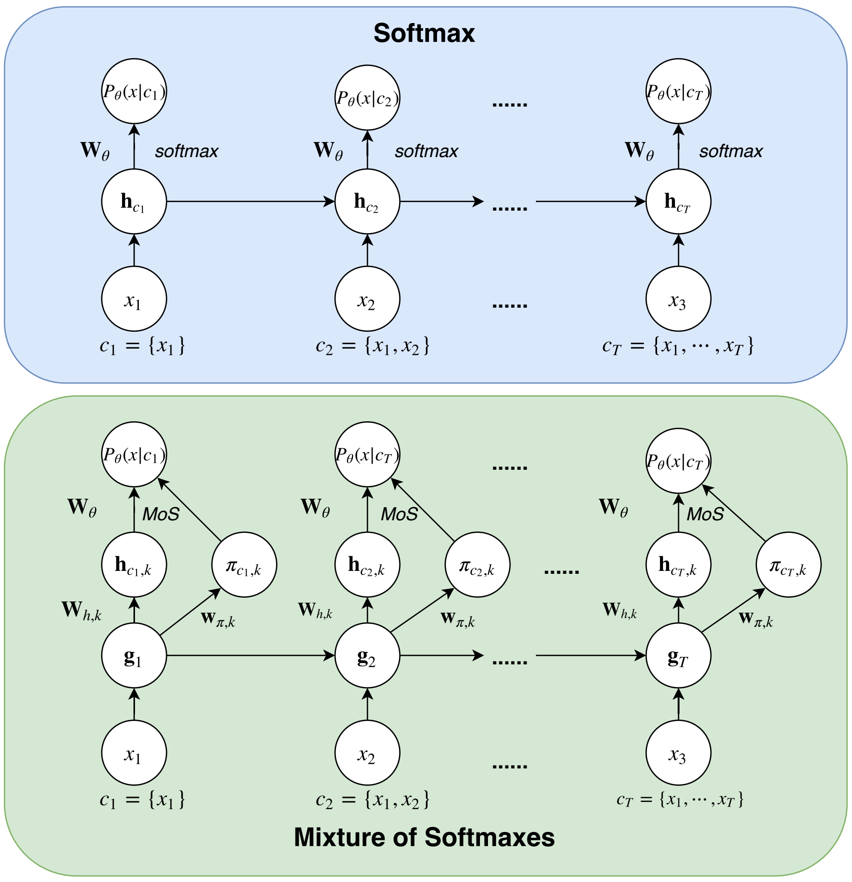


Figure 6.1: RNN structures with Softmax and MoS.

However, the drawback of MoS is that more Softmax functions also bring a large number of parameters, which will lead the model to overfit again. In fact, since MoS is able to increase the expressiveness of the models, we can reduce the number of hidden units to compensate for the parameters that MoS brings. That is to say, a smaller model with MoS could have better expressiveness and generalization than a larger model with standard Softmax layer, which will be verified in Section 6.3.

6.3 Experiments and Results

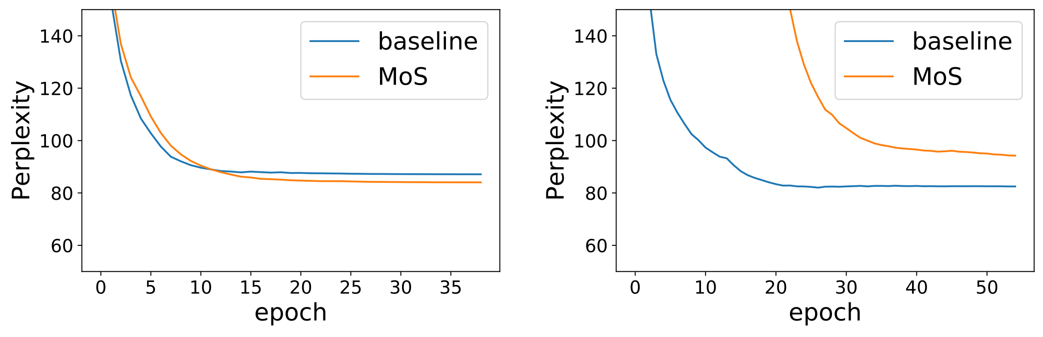
The experiments are designed to explore the effectiveness of MoS on NNLMs. Three experiments are carried out. At first, we explore how MoS works on our baseline models with and without tuning hyperparameters. Then we shows the how the number of Softmax components in MoS influences perplexities and speeds. Finally, we compare MoS models with baseline models controlling variables.

6.3.1 Compared with Baseline

At first, we apply MoS directly on our small, medium and large models as discussed in Chapter 5 with basic and block LSTM cells. All hyperparameters are fixed to be the same as the baselines, and the additional hyperparameter, the number of Softmax exponents, is set as 15 for all models. Only the results of the better models between the basic and block LSTM cells are shown in Table 6.1.

model	**small**		**medium**		**large**	
	baseline	MoS	baseline	MoS	baseline	MoS
	block	block	basic	basic	basic	block
training ppl.	40.064	29.020	45.476	53.967	37.782	70.265
validation ppl.	118.463	127.718	87.127	84.047	82.486	94.249
test ppl.	113.304	121.748	83.179	80.441	78.195	90.138
training speed (wps)	17770	3760	8850	1980	2860	930
total time (h)	0.273	0.998	1.245	5.321	5.133	15.746
#parameters (M)	4.652	5.258	19.775	26.132	66.022	99.817
peak GPU RAM usage (GB)	0.065	0.917	0.940	1.773	4.830	5.080

Table 6.1: Performance of MoS models compared with baseline models, where the better one between basic and block LSTM is shown.



(a) validation ppl. of medium models (b) validation ppl. of large models

Figure 6.2: The validation perplexities of the medium and large models with and without MoS.

It can be seen that MoS just improves perplexities in the medium models, taking more than 4 times longer. In the small and large models, MoS makes perplexities worse, although more time is spent. The hyperparameter configurations are to blame for these results, because they are well-tuned in the baseline models and may not suitable for the models with very different structures. The medium MoS models gets the best results among the MoS, while the results of large baseline models are better than the small and medium ones. This implies that, when applying MoS, we should decrease the model size to compensate for the parameters brought by MoS, as mentioned in Section 6.2.

MoS increases expressiveness of models, so in theory training perplexities should be lower, but the reality is on the contrary for the medium and large models. From Figure 6.2, it can be seen that the MoS models have slower convergence rate than the baseline models, which is the reason why MoS does not perform as good as we expect. Hence, they need new learning rate schedules to achieve their potential.

We apply two strategies to tune hyperparameters of MoS models. The first one is just to select better learning rate schedules for the medium models. We train the models with the original learning rate, and choose the epochs where validation perplexities achieve their minimum. That is to say, after those epochs, the models tend to overfit. Then we roll back to the chosen epochs, train the models with a learning rate decay of 0.8 at the end of each epoch, and stop at the epoch with the lowest validation perplexity. The second strategy is to optimize all hyperparameters to achieve a best performance of MoS models. This task is hard because the dimensionality of the hyperparameters is high, so tuning them by grid search is too expensive and not feasible. Therefore, we use random search (Bergstra and Bengio, 2012) to optimize the hyperparameters. To be explicit, we set the ranges of all hyperparameters except the original learning rate, then randomly generate many hyperparameter configurations in the ranges. We fix the original learning rate as 1, which is the same as in the baseline models. Then the MoS models with these configurations are trained with the original learning rate for 20 epochs, and the lowest validation perplexity for each model is recorded. Then we choose the model with lowest validation perplexity among these models, and train it longer with tuning its learning rate schedule by the first strategy. The ranges of the hyperparmeters are shown in Appendix A.2. Random search is more efficient than grid search when dimensionality of hyperparameters is high. The computational complexity is exponential to dimensionality of hyperparameters. However, sometimes not all the hyperparameters are essential to impact the results, which is called low effective dimensionality (Bergstra and Bengio, 2012). In this case, random search generates much more values than grid search on the effective dimensions within the same time, because random search generates values on each dimension randomly while grid search duplicates the same values on each dimension. The results of the MoS models with well-tuned hyperparameters are shown in Table 6.2, and the details of the hyperparameter configurations are in Appendix A.2.

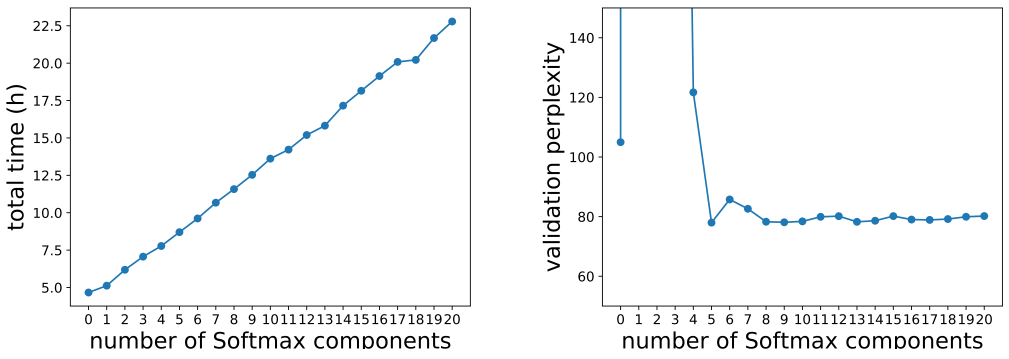
model	**#epochs**	**val ppl.**	**test ppl.**	**speed** (wps)	**total time** (h)	**#params** (M)
large baseline	55	82.486	78.195	2860	5.133	66.022
medium MoS	78	81.932	77.264	1990	10.465	26.132
random search	100	78.618	73.565	1550	17.159	48.826

Table 6.2: Performance of MoS models with well-tuned hyperparameters compared with the baseline model of the lowest perplexity. The large baseline model uses basic LSTM cell, the medium MoS model uses the block LSTM cell and a tuned learning rate schedule, and the MoS model with random search has 14 Softmax components.

Table 6.2 shows that two MoS models have less parameters and achieve lower validation perplexities than the baseline model, which proves that MoS can increase the capacity of the models and less parameters relieve overfitting. According to Formula 6.2, the extra parameters brought by MoS are $\mathbf{w}_{\pi,\mathbf{k}}$ and $\mathbf{W}_{\mathbf{h},\mathbf{k}}$, whose dimensions are not related to the vocabulary size, so the increasing of the number of parameters is not extremely high. However, MoS computes Softmax functions K times, which significantly increase the amount of computations. That is why although the sizes of MoS models are smaller than the baseline model, the training speeds are lower. Also, MoS models need more training epochs to achieve the optima.

6.3.2 Influence of the Number of Softmaxes

From Formula 6.2, more Softmaxes components bring more parameters and computations. We proved that MoS models can achieve lower perplexities, while spending much more time. Decreasing the number of Softmaxes could improve efficiency of MoS models. Therefore, we explore how the number of Softmaxes influences perplexity and speed of MoS models. We use the MoS model with random search and fix all hyperparameters except the number of hyperparameters, K, and make a comparison. We set K in the range [0, 20], and carry out experiments with all K, where $K = 0$ means using the standard Softmax function.



(a) total time vs #Softmaxes (b) perplexity vs #Softmaxes

Figure 6.3: The influence of the number of Softmax compoments on total times and perplexities.

From Figure 6.3(a), we could see that the training time is proportional to K, and the relation between them is almost linear, because more softmaxes lead to more parameters and computational complexity. However, according to Figure 6.3(b), K does not have linear relation with perplexity. Actually, when K is no less than 4, the perplexity fluctuates in a small range. This implies that the increase of the number of Softmax component cannot improve the perplexity. As is mentioned in Section 6.2, the advantage of MoS is breaking the “linearity” of Softmax functions, which can be done by a few Softmax components. Hence, more Softmax components do not bring more advantages. When K is 1 or 2, the perplexity explodes and does not converge, because these models are very sensitive to the learning rate, and the current learning rate seems to be too high for them. Lower learning rate may lead the models to a better result. When K is 3, the perplexity converge to a large number, 531.063. In addition, the model with $K = 1$ is not equal to the one with $K = 0$, because the former has one more additional dense layer before the output layer. This is different from the argument in Yang et al. (2018). These experiments can also be regarded as an ablation study, which shows that models with MoS have lower perplexities than the model with standard Softmax function.

In random search, the best model has 14 Softmax components, while we find the model with 5 Softmax components has lower perplexity (77.981 on validation set and 73.898 on test set) and higher speed. In terms of the perplexity, the latter model is also the best MoS model so far. We call it BM (best MoS) model. BM models will also be used in the following experiments.

The BM model is also trained[1] using MXNet (Chen et al., 2015), which is also a symbolic machine learning framework, for five epochs. The average time per epoch taken by the MXNet model is 243.6 seconds, while in TensorFlow the BM model takes 306.7 seconds per epoch. This implies that MXNet is more efficient than TensorFlow in training the BM model.

6.3.3 Further Analysis with Variables Controlled

MoS improves perplexities of models, however it makes convergence slower. We want to know if MoS models can achieve the same or better perplexity within the same time of baseline models. We experiment on BM model compared with the large baseline model controlling training times or validation perplexities, and the results are shown in Table 6.3.

model	**val pp.**	**speed** (wps)	**training time** (h)	**#param** (M)
large baseline	82.486	2860	5.050	66.022
BM	83.956	3060	5.076	41.520
	82.342		5.248	
	77.981		8.606	

Table 6.3: Performance of the BM model compared with the large baseline model, where wps means words per second.

Table 6.3 tells us the BM model is also smaller than the large baseline models, and the training speed of the former is a little higher than the latter. However, the convergence rate of the BM model is slower than the large baseline model. For the similar training time, the validation perplexity of the BM model is slightly higher (about 1.5) than the large baseline model, and the BM model achieves similar or slightly better perplexity by spending a little more time (about 0.2 hour). However, compared with the large baseline model, the BM model can make a clear progress in perplexity (about 4.5) by training 3.55 hours longer. In conclusion, MoS is suitable for that tasks where perplexity is very important; if long training time is not acceptable, models with standard Softmax function are the better choices.

[1]By Xin Chen, one of my groupmates, in his experiments.

Chapter 7

Speeding up Training

MoS achieves lower perplexities while making training slower. In this chapter, we attempt to speed up training by using several tricks.

7.1 GPU Parallelization

All previous experiments are run on one GPU. In this section, we parallelize models on several GPUs to speed up computations. To be explicit, in each training iteration, we separate the input batch to several GPUs to compute the forward and backward propagations, then add up the gradients from all GPUs to update the parameters. In TensorFlow, this process is presented as parallelizing computation graphs to several GPUs. Since the data chunk is split between several GPUs, the training speed could be increased. However, GPUs need to communicate with each other at the end of every iteration. There are so many iterations in the whole training time, so the communications are an overhead, which could clearly influence training speeds. Furthermore, there is a buckets effect at the end of each iteration, when all GPUs who have done the work faster need to wait for the slowest GPU.

GPU parallelization may increase the whole batch size if we fix the batch size on each GPU. Hence, the whole batch sizes are different for different numbers of GPUs. According to Figure 4.2, increasing batch size means less iterations per epoch, which is the main reason why GPU parallelization can speed up training. However, different batch sizes for different numbers of GPUs could lead to different perplexities.

7.2 Weight Tying

A effective way to speed up training is to reducing the number of parameters. Press and Wolf (2016) proposed weight tying to do this, which worked well in the large baseline models, as shown in their paper. In the traditional concepts, the first dense layer who encodes high-dimensional input word vectors to low-dimensional hidden vectors works as a word embedding. However, the function of the last layer before the Softmax function is decoding the low-dimensional word vectors to high-dimensional output word vectors, which is similar to the inputs encoding matrices. Therefore, we also call this matrix an embedding matrix. Since these two matrices do the similar thing, we tie their weights. To be explicit, we makes the output embedding matrix is exactly the transpose of the input embedding matrix, where the weights are shared.

In the MoS model, the decoding matrix is $\mathbf{W}_\theta$ in Formula 6.3, where the i^{th} row is $\mathbf{w}_{x_i}$ in Formula 6.2. Since for all softmax exponents $\mathbf{w}_{x_i}$ are the same, so $\mathbf{W}_\theta$ is the same. Therefore, weight tying can also be applied on MoS models, making $\mathbf{W}_\theta$ be the transpose of the inputs embedding matrix.

Since the size of vocabulary is usually much larger than the size of hidden layers, these two matrices have a great amount of parameters. Hence, weight tying significantly decreases the number of parameters, and speeds up training. However, weight tying makes models less flexible. In fact, the training perplexities of untied models should be no higher than of tied models. On the other hand, less parameters may lead to better generalization of tied models.

7.3 Weight Matrix Factorization

The other trick we use to reduce the number of parameters is to factorize the weight matrix in the Softmax layer. This thought is inspired by Kuchaiev and Ginsburg (2017). This paper proposed to factorize the LSTM weight matrix. According to Formula 4.1, in each LSTM cell, the size of weight matrix W is $2P \times 4Q$, where P is the size of inputs and Q is the size of hidden units. If W is factorized as $W_1 \times W_2$, where the sizes of W_1 and W_2 are $2P \times r$ and $r \times 4Q$, the number of parameters in this layer decreases when $r < \frac{4PQ}{P+2Q}$. Formula 4.1 becomes as follows.

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W_1 W_2 \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \tag{7.1}$$

However, this trick will reduce the capacity of the model, because the rank of $W = W_1 W_2$ is no more than r which is smaller than the original rank. Therefore, this trick may significantly hurt the perplexity. To deal with this problem, we apply matrix factorization on the weight matrix in the Softmax layer instead of the weight matrices in the LSTM cells, i.e. $\mathbf{W}_\theta = \mathbf{W}_{\theta,1}\mathbf{W}_{\theta,2}$. If we apply this trick on RNN models with standard Softmax layers, the approximated logarithm probability distribution matrix becomes $\mathbf{A}' = \mathbf{H}_\theta(\mathbf{W}_{\theta,1}\mathbf{W}_{\theta,2})^T$, where the rank of $\mathbf{W}_{\theta,1}\mathbf{W}_{\theta,2}$ will influence capacity of models. However, this is not a problem in MoS models, where Formula 6.3 becomes as follows.

$$\hat{\mathbf{A}} = \log \sum_{k=1}^{K} \Pi_k \exp(\mathbf{H}_{\theta,k}(\mathbf{W}_{\theta,1}\mathbf{W}_{\theta,2})^T) \tag{7.2}$$

Since $\hat{\mathbf{A}}$ is not constrained by the rank of $\mathbf{W}_{\theta,1}\mathbf{W}_{\theta,2}$, capacity of MoS models will not be reduced by the decreasing of the rank.

7.4 Gradient Clipping

In order to simplify the process of learning rate schedule optimization, the original learning rate is fixed to 1 when randomly searching for hyperparameters. In fact, a too large learning rate will lead to divergence of perplexities, while too small learning rate will slow down the convergence rate. In the previous model, we also use gradient clipping to avoid gradient explosion. That is to say, after each back propagation, if the square sum of the gradients S is larger than the threshold S_{max} we set, each gradient will be multiplied by $\frac{S_{max}}{S}$. This guarantees that the square sum of gradients for each back propagation is no more than the threshold, so as to prevent gradient explosion.

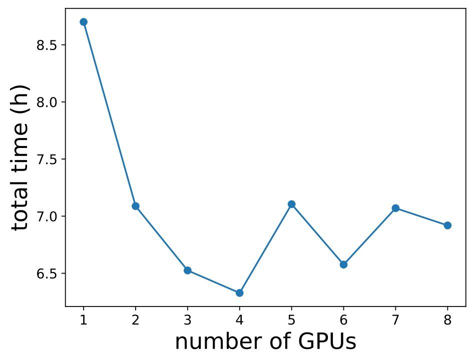
In the previous experiments, we use a large threshold, which is around 5 to 10, just to prevent gradient explosion, while in this section, we use small threshold with large learning rate to speed up training. Gradient clipping can effectively avoid too large gradients, so a larger learning rate may be allowed when a more strict gradient clipping applies. When a larger learning rate is applied, the model may converge faster, and less training epochs are needed. Therefore, we could speed up training by choosing a larger learning rate and a proper gradient clipping, without leading perplexities to diverge.

7.5 Experiments and Results

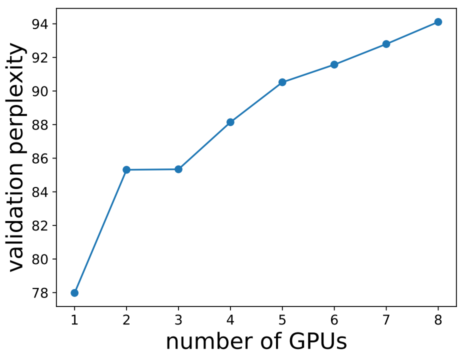
The following series of experiments are designed for verifying the effectiveness of the speed optimization tricks introduced in this chapter.

7.5.1 Influence of the Number of GPUs

We explore the influence of the number of GPUs on the BM model. All hyperparameters are fixed except batch size and the number of GPUs. The batch size is fixed on each GPU. That is to say, we allocate the same number of batches on each GPU. If the number of GPUs increases, the whole batch size increases to guarantee that the number of batches dealt with by each GPU is the same. According to Figure 4.2, since the length of sequence is fixed, the number of iterations in per epoch decreases with an increase in batch size. The number of total epochs is fixed, too. The experiments are carried out by 1 to 8 NVIDIA GTX1060 GPUs. The results are shown in Figure 7.1.



(a) total time vs #GPUs



(b) perplexity vs #GPUs

Figure 7.1: The influence of the number of GPUs on total times and perplexities.

Figure 7.1(a) reveals that when the number of GPUs is no more than 4, increasing the number of GPUs significantly improve the speed of the BM model. However, when the number of GPUs exceeds 4, the total time fluctuates and is not stable. The reason is more GPUs bring heavier overhead of communications, which impact the speed. From Figure 7.1(b), one can see that validation perplexity increases with the number of

GPUs increasing. The main reason is that the hyperparameters are fine-tuned on single GPU, so the original batch size is well compatible with other hyperparameters. Hence, simply increasing the batch size will negatively impact the perplexity. A solution is to tune hyperparameters again on multi-GPU, fixing the batch size per GPU.

We also explore whether the perplexities are identical if we keep the whole batch size the same. The BM model is also used as the baseline here, while the new model with 2 GPUs allocates half of the batch size per GPU to keep the whole batch size. The results in shown in Table 7.1, where the two results are similar and the slight difference comes from different weights initializations. Since the number of iterations per epoch is the same and the communications between 2 GPUs are a heavy overhead, the model with 2 GPUs spend around double time compared with the model with single GPU.

#GPUs	**val pp.**	**speed** (wps)	**training time** (h)
1	77.981	3060	8.700
2	80.960	1620	16.461

Table 7.1: Performance of the BM model with one or two GPUs, keeping the whole batch size.

From Figure 7.1, a good tradeoff between perplexity and total time is 3 or 4 GPUs. However, these models spend more time than the large baseline model to get a worse perplexity, which is not desired. Therefore, simply increasing the number of GPUs cannot solve the problem of speed brought by MoS, and further hyperparameter optimization is necessary.

7.5.2 Operations on Weight Matrices

We apply weight tying and weight matrix factorization, discussed in section 7.2 and 7.3, on the medium MoS models, compared with the medium baseline and untied MoS models. All hyperparameters are fixed. For the model with weight matrix factorization, an extra hyperparameter is the low rank r for $\mathbf{W}_{\theta,1,2P\times r}$ and $\mathbf{W}_{\theta,2,r\times 4Q}$, which is set as 256. Only the weight matrix of MoS layer are factorized. The experimental results are shown in Table 7.2.

It is clear that both weight tying and weight matrix factorization reduce the number of parameters and speed up training significantly. The training perplexity of the baseline model is the lowest among four models. As is discussed in Section 6.3.1, MoS models converge slower than baseline models. The total number of epochs is fixed and suitable for baseline, so the MoS medium model needs more epochs to achieve lower training perplexity. Weight tying and weight matrix factorization both reduce the capacity of the models, so the training perplexities of these two models are higher than the untied MoS model. On the other hand, MoS improves the validation perplexity as compared to the baseline model, although MoS increases the number of parameters. Weight tying further low down the validation perplexity to 80.336. The progress of generalization comes from less parameters. However, the MoS model with weight matrix factorization does not perform as well as we expect. The validation perplexity of this model is even higher than the baseline model, while it spends more time on training than the baseline model. Hence, this model does not have any advantages compared with the baseline model. It is noteworthy that the hyperparameter configuration for these four models are tuned on the baseline model, so it may not suit other models so well. If we further tune the hyperparameters on the other three models, respectively, their performance may be improved.

model	**standard**	**MoS**	**weight tying + MoS**	**factorization + MoS**
training ppl.	45.476	53.967	61.584	63.305
validation ppl.	87.127	84.047	80.336	89.882
test ppl.	83.179	80.441	77.072	86.005
training speed (wps)	8850	1980	2590	2790
total time (h)	1.245	5.321	4.082	3.781
#parameters (M)	19.775	26.132	17.503	20.240
#GPU Usage (GB)	0.940	1.773	1.200	1.217

Table 7.2: Performance of medium baseline and MoS models with and without speed optimization tricks.

7.5.3 Learning Rate Optimization with Gradient Clipping

As discussed in Section 7.4, a small gradient clipping threshold and a large learning rate may speed up training. We re-optimize the learning rate schedule with strict gradient clipping on the BM model, in order to make the model converge faster with a larger learning rate. The maximum gradient threshold decreases from 7.2 for the BM

model	**first stage #epochs**	**whole stage #epochs**	**val pp.**	**speed** (wps)	**training time** (h)
large baseline	14	55	82.486	2860	5.050
BM	51	100	77.981	3060	8.606
BMLG	34	60	79.440	3800	4.175

Table 7.3: Performance of BM model and BMLG model.

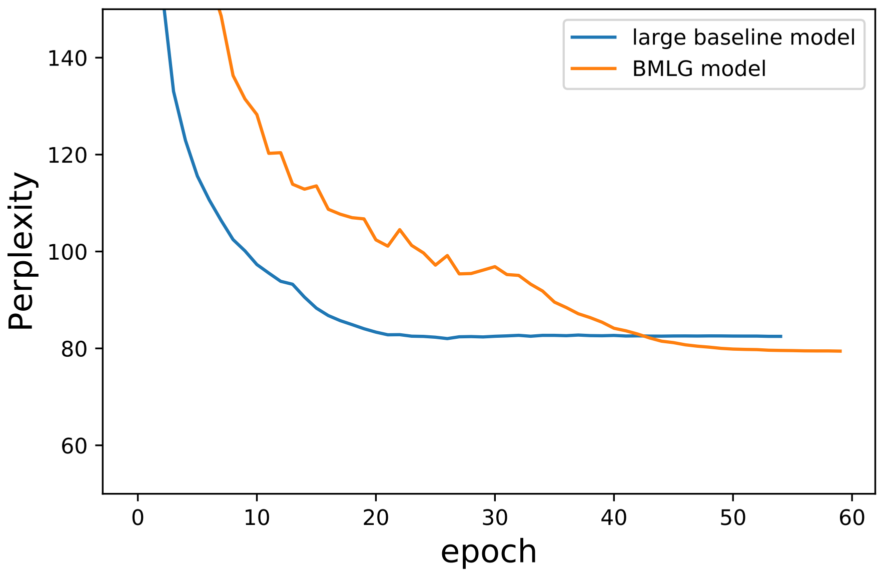


Figure 7.2: The validation perplexities of the large baseline model and the BMLG model.

model to 0.3 for the new model. Then the original learning rate increases from 1 for the BM model to 20 for the new model. All other hyperparameters are fixed.

The new model is trained with the new original learning rate for the first stage until overfitting occurs. Then the model is trained with learning rate decay (0.8) for the second stage. The new BM model with optimized learning rate (L) and gradient threshold (G) is called the BMLG model. The comparison with the BM model with the BMLG model are shown in Table 7.3.

From Table 7.3, we can see that the BMLG model has a higher convergence rate than the BM model and needs less epochs to train. Hence, the total time of the BMLG model is much shorter than the BM model. Furthermore, the average training speed for the BMLG model is also higher than the BM model. The BMLG model achieves a slightly higher validation perplexity with the BM model in a much shorter time, which is a great progress. The new learning rate schedule with gradient clipping improves the slow convergence rate of MoS models.

We can also see in Table 7.3 that the BMLG model has both better perplexity and speed than the large baseline model. This implies BMLG could be the first choice for the users who value perplexity or speed. From Figure 7.2, the large baseline model converges faster than the BMLG model in the first 40 epochs, while the BMLG model achieves lower perplexity after that.

Chapter 8

Comparison with PyTorch

8.1 Introduction of PyTorch

PyTorch (Paszke et al., 2017) is an imperative machine learning framework. Different from static graphs of TensorFlow, PyTorch is an dynamic framework, where the functions run immediately when they are encountered. Hence, PyTorch does not have to build a static graph and reuse it again and again in iterations. Instead, PyTorch uses and replays a tape recorder to build networks. That is to say, the operations will be recorded when they are run in forward propagation, and the gradients will be automatically computed in the reversed order in back propagation, which is called reverse-mode auto-differentiation. This makes it easy to change the networks. The grammar of PyTorch is more straightforward than TensorFlow. We do not have to separately build static graphs and run them, which makes the codes neat. Users can build layers by pure Python. PyTorch also has good extensibility, where other Python package, including Numpy, Scipy and Scikit-learn, can be used in PyTorch framework.

8.2 Implementations using PyTorch

Dynamic frameworks can deal with variable lengths of inputs better, whereas static frameworks only allows the inputs have the same length. Hence, PyTorch is more flexible in building NNLMs than TensorFlow, because the lengths of sentences may differ. The model defined in PyTorch is much easier than in TensorFlow, and the length of the code is shorter. It is easier for PyTorch to parallelize models to multiple GPUs. We only need to add one more statement, `model = nn.DataParallel(model)`, in our code. However, in TensorFlow, we need to export all the operations to an meta-

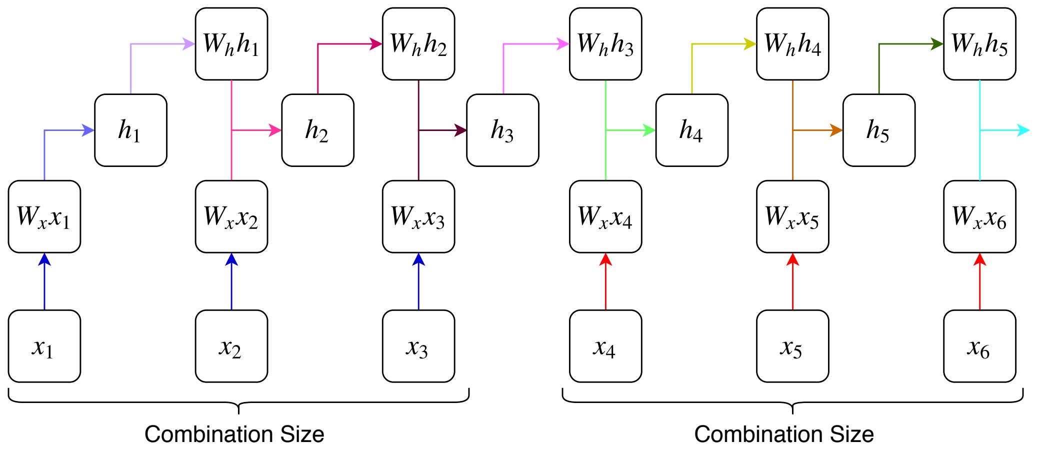


Figure 8.1: Fused computations for two streams in CUDNN LSTM. The arrows of the same color means these computation occurs simultaneously.

graph, and parallelize the metagraph to multiple GPUs, which is a complicated process. We encounter a same problem in implementing NNLMs in TensorFlow and PyTorch. Some built-in functions in these frameworks are invalid in many versions, and different versions may have different ways and functions to implement the same thing. This will reduce the transferability of the code.

PyTorch uses CUDNN mode to implement RNN by default on GPUs (only by `model.cuda()`). TensorFlow also provides CUDNN mode. However, the implementation of CUDNN LSTM (`tf.contrib.cudnn_rnn.CudnnLSTM`) is very different from basic or block LSTM cells, which increases the difficulty of using it. CUDNN is a good way to optimize models on GPUs. For LSTM models, CUDNN helps to change the order of computations and accelerate training[1]. To achieve GPU peak floating-point throughput, combining computations in a large matrix-matrix multiplication is a good idea. Formula 4.1 actually has 8 matrix-matrix multiplications from two inputs (x_t and h_{t-1}) to four outputs (i, f, o and g gates). The basic and block LSTM cells combine the 8 matrix-matrix multiplication to one by stack the inputs, which increase the throughput of GPUs. CUDNN does this in a different way. The computations can be divided into two streams. One is the computation on the inputs from the last layer, x_t; the other is the computations on the inputs from the last time stamp, h_{t-1}. The computations in the first stream can be combined into a larger matrix-matrix multiplication because we have all inputs. This can further increase the throughput of GPUs. However, the computations in the second stream cannot occur simultaneously, because the

[1]https://devblogs.nvidia.com/optimizing-recurrent-neural-networks-cudnn-5/

computations at the latter time stamp have to wait for the results of the computations at the former time stamps. Combining the whole computations in the first stream together will prevent the overlapped computations with the section stream. Therefore, we choose a combination size and only combine part of inputs at once. This process is shown in Figure 8.1. Furthermore, the computations in layer-level can be optimized. In the traditional computation order, computations at one time step occur only after all layers at the previous time stamp done. CUDNN fuses these computations, where in all layers the computations occur immediately when all inputs are available. That is to say, the computations can spread from the start point, which is shown in Figure 8.2. In the back propagation, computations occur in the similar way. These optimizations are automatically done by CUDNN LSTM.

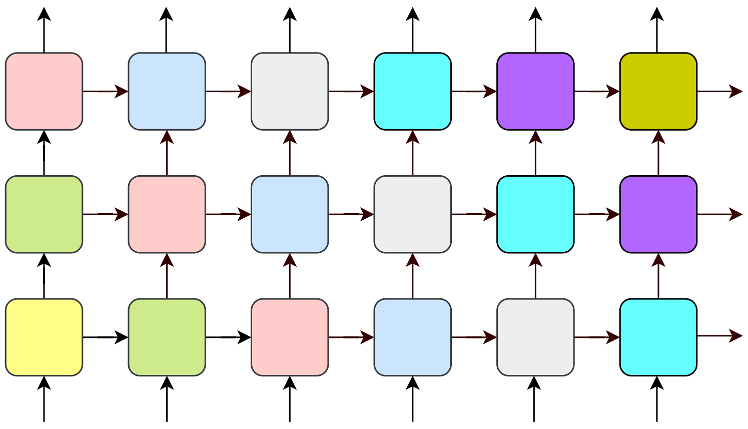


Figure 8.2: Fused computations for layers in CUDNN LSTM. The blocks of the same color means these computation occurs simultaneously.

8.3 Experiments and Results

We implement the BMLG model by PyTorch 0.1.12 one one NVIDIA GTX960 GPU, and the codes are modified from an implementation of standard LSTM[2]. No hyper-parameters are changed. The performances of the BMLG models in TensorFlow and PyTorch and shown in Table 8.1 and Figure 8.3.

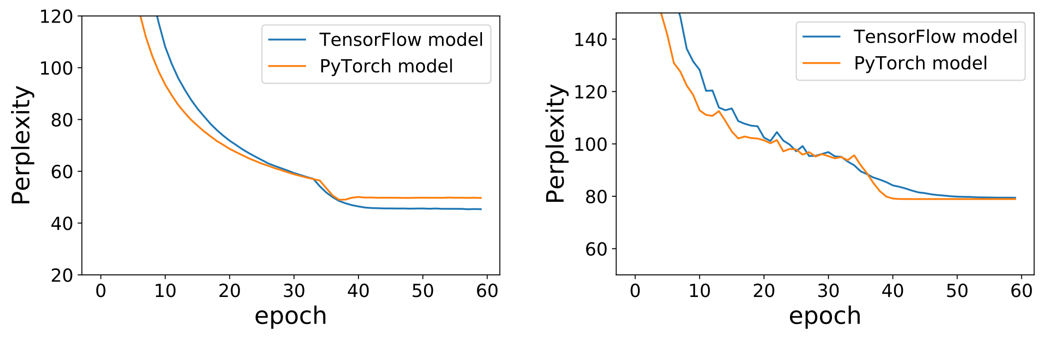
From Table 8.1, we can see that the PyTorch model has higher training speed than TensorFlow, which is from the contribution of CUDNN LSTM. The perplexity of the PyTorch model is also slightly lower than the TensorFlow model. We further tune the threshold of gradient clipping in PyTorch, and makes a little progress on perplexity

[2]https://github.com/deeplearningathome/pytorch-language-model

model	**#epochs**	**val pp.**	**speed** (wps)	**training time** (h)
TensorFlow	60	79.440	3800	4.175
PyTorch	60	78.943	4860	3.258
PyTorch (tuned)	45	78.535	4860	2.469

Table 8.1: Performance of the BMLG models in TensorFlow and PyTorch.

and increase the convergence rate of the model. If the threshold decreases to 2.5, the PyTorch model achieves 78.535 perplexity, while converges in 45 epochs (the first stage is also 34 epochs). Though the training speed does not change, less epochs lead to shorter total time.



(a) training perplexity of BMLG models (b) validation perplexity of BMLG models

Figure 8.3: The training and validation perplexities of the BMLG models in TensorFlow and PyTorch (not tuned).

Figure 8.3 shows that PyTorch model (not tuned) converges faster in terms of both training and validation perplexities. However, TensorFlow model achieves clearly lower training perplexity than PyTorch model, which implies the former has stronger expressiveness. The two models finally achieve similar validation perplexities, which implies that they have similar generalization.

From the comparison, we could say PyTorch has advantages in implementing the BMLG model than TensorFlow, because the PyTorch model has significantly higher speed and slightly lower validation perplexity. However, if the CUDNN LSTM in TensorFlow is also applied when implementing the BMLG model, the speed of TensorFlow model may be improved.

Chapter 9

Future Work

The best model we implement is the BMLG model, which has the best perplexity and training speed among these models. However, the perplexity of the BMLG is not state-of-the-art. That is to say, many advanced techniques that were only implemented in PyTorch can be applied to optimize the TensorFlow language models.

First of all, as mentioned in Section 7.2 and Section 7.3, weight tying and weight matrix factorization can be used to optimized speeds of MoS models, while further hyperparameter optimization is necessary because these tricks changes the architectures of the original models. Hyperparameter optimization can also be done by random search (Bergstra and Bengio, 2012), while a more advanced method, Bayesian Optimization, can also be used. Bayesian Optimization (Snoek et al., 2012) builds the function between the hyperparameters and the loss by Gaussian processes, and uses acquisition functions to find the next hyperparameter configuration which has highest probability to improve the loss. Then the results of the new configuration will be modelled by Gaussian processes again along with the previous configurations, in order to find the next configuration. Bayesian Optimization makes a good use of the informations of previous hyperparameter configurations to find the next ones, instead of iteratively searching for hyperparameter configurations blindly. However, the Gaussian processes have larger computational complexity.

Secondly, many advanced techniques mentioned in Section 2.2 can be applied on NNLMs in TensorFlow. One technique worth to try is variational dropout (Gal and Ghahramani, 2015), which applies dropout on both recurrent and non-recurrent connections while keeps the dropout mask the same at every time stamps. This technique regularizes the recurrent connections, thus further reduces overfitting. In TensorFlow, the basic and block LSTM cells provide variational dropout as a parameter. However, the variational dropout has to be implemented outside LSTM cells, such as in the embedding matrix and the latent layer in MoS. Another advanced model is called AWD-LSTM (Merity et al., 2017). Yang et al. (2018) applied MoS on AWD-LSTM in their paper using PyTorch and achieved state-of-the-art perplexity, while in our project, we just apply MoS on the standard LSTM with non-recurrent dropout, and thus get higher perplexity. AWD-LSTM uses two main tricks. One is weight dropout as mentioned in Section 2.2, and the other is average SGD (ASGD). Merity et al. (2017) mentioned it is easy to implement weight dropout because it does not change the architecture of LSTM cells. This is not true in TensorFlow, because the weights of LSTM cells are defined in the cells. We need to make new LSTM cells which are inherited from the existing LSTM cells and change the weights defined inside. ASGD is a new optimizer to TensorFlow. All existing optimizers are written in C++ and encapsulated by TensorFlow operations. Hence, the best way to implement ASGD is to add a new TensorFlow operation using C++, which is not easy.

Finally, the CUDNN LSTM cell in TensorFlow `tf.contrib.cudnn_rnn.CudnnLSTM` may further speed up the models. As mentioned in Section 8.2, the CUDNN LSTM cell computes loss and gradients in a different way and makes computations more efficient. The CUDNN LSTM cell also uses a different way to define and store their weights. The weights are defined outside the cell and converted into another form in the cell, which makes use of economic GPU memory. AWD-LSTM proposes to separate the dimensions of the last hidden layer and the previous hidden layers. This can be done easily in the basic and block LSTM cells, because they define a single layer at once and wrap all the layers by `tf.contrib.rnn.MultiRNNCell`. However, the CUDNN LSTM cell defines all layers together and does not support separate dimensions. The CUDNN Cell also does not support variational dropout. It is very hard to implement these tricks in CUDNN LSTM cells.

Chapter 10

Conclusion

We verified the performance of TensorFlow on neural network language models (NNLMs). In baseline experiments, we modelled language by long short-term memory (LSTM) recurrent neural networks (RNN) with non-recurrent dropout, and trained three models of different sizes. The larger models had lower perplexities but slower speeds. Then Mixture of softmaxes (MoS) was applied to improve the perplexity of the baseline models. Keeping the same hyperparameters as the baseline models, MoS only improved the perplexity of the medium baseline model, while increased the perplexity of the small and large baseline models. This implies that the hyperparameters needed to be tuned. MoS adds more parameters and increases the expressiveness of the models, so the best MoS (BM) model after tuning hyperparameters has smaller size compared with the large baseline model, and has faster training speed and lower perplexity. However, the convergence rate of the BM model is slower than the large baseline model. Hence, the BM model spends more training epochs to get a lower perplexity.

To speed up training, several tricks were applied. We parallelized the BM model on multiple GPUs, which increased training speeds while leading to higher perplexities due to the increase of batch sizes. Then we applied weight tying and weight matrix factorization on the medium MoS model to reduce the number of parameters to speed up training and reduce overfitting. Weight tying also improved validation perplexity of the medium MoS model. Finally, we used a strict gradient clipping strategy with a large learning rate, which increased the convergence rate of the BM model. We called the new model the BMLG model.

We compare the BMLG models implemented in both TensorFlow and PyTorch. The two models have similar perplexities, while the latter is easier to implement and has higher speed thanks to the CUDNN LSTM cell.

Appendix A

Hyperparameter Configurations

The hyperparameters of TensorFlow neural language models in this project are as follows:

- **Initial scale**: the range of weights initialization by uniform distribution.
- **Learning rate**: the original learning rate from the beginning of training.
- **Maximum norm**: the threshold for gradient clipping.
- **Number of layers**.
- **Hidden size**: the number of hidden units.
- **Batch size**.
- **Length of sequences**.
- **Stage 1**: the number of epochs in the first training stage.
- **Stage 2**: the number of epochs in the whole training stage.
- **Keeping probability**: the probability of keeping a unit in dropout layers.
- **Learning rate decay**: the decay rate in the second training stage.
- **RNN mode**: build LSTM using basic or block cells.
- **Number of GPUs**.
- **Number of components**: the number of Softmaxes functions in MoS layer. Only for MoS models.

A.1 Baseline

Hyperparameters	**small**	**medium**	**large**
Initial scale	0.1	0.05	0.04
Learning rate	1.0	1.0	1.0
Maximum norm	5.0	5.0	10
Number of layers	2	2	2
Hidden size	200	650	1500
Batch size	20	20	20
Length of sequences	20	35	35
Stage 1 & 2	4 & 13	6 & 39	14 & 55
Keeping probability	1.0	0.5	0.35
Learning rate decay	0.5	0.8	$\frac{1}{1.15}$

Table A.1: The hyperparameters for baseline models.

A.2 Random Search for MoS

Hyperparameters	**range for random search**	**results**
Initial scale	(0, 0.1)	0.02
Learning rate	1.0	1.0
Maximum norm	[5.0, 10.0]	7.2919
Number of layers	[1, 3]	3
Hidden size	[500, 1500]	900
Batch size	[10, 30]	28
Length of sequences	[20, 50]	40
Stage 1 & 2	20 & 20	51 & 100
Keeping probability	(0.3, 0.5)	0.4411
Learning rate decay	1.0	0.8
RNN mode	basic or block	block
Number of components	[2, 20]	14

Table A.2: The range of hyperparameters in random search for MoS models, where batch size $\times$ length of sequences $\times$ hidden size $\leq 1.2 \times 10^6$, and the results.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I. J., Harp, A., Irving, G., Isard, M., Jia, Y., Józefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D. G., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P. A., Vanhoucke, V., Vasudevan, V., Viégas, F. B., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467.

Al-Rfou, R., Alain, G., Almahairi, A., Angermüller, C., Bahdanau, D., Ballas, N., Bastien, F., Bayer, J., Belikov, A., Belopolsky, A., Bengio, Y., Bergeron, A., Bergstra, J., Bisson, V., Snyder, J. B., Bouchard, N., Boulanger-Lewandowski, N., Bouthillier, X., de Brébisson, A., Breuleux, O., Carrier, P. L., Cho, K., Chorowski, J., Christiano, P. F., Cooijmans, T., Côté, M., Côté, M., Courville, A. C., Dauphin, Y. N., Delalleau, O., Demouth, J., Desjardins, G., Dieleman, S., Dinh, L., Ducoffe, M., Dumoulin, V., Kahou, S. E., Erhan, D., Fan, Z., Firat, O., Germain, M., Glorot, X., Goodfellow, I. J., Graham, M., Gülçehre, Ç., Hamel, P., Harlouchet, I., Heng, J., Hidasi, B., Honari, S., Jain, A., Jean, S., Jia, K., Korobov, M., Kulkarni, V., Lamb, A., Lamblin, P., Larsen, E., Laurent, C., Lee, S., Lefrançois, S., Lemieux, S., Léonard, N., Lin, Z., Livezey, J. A., Lorenz, C., Lowin, J., Ma, Q., Manzagol, P., Mastropietro, O., McGibbon, R., Memisevic, R., van Merriënboer, B., Michalski, V., Mirza, M., Orlandi, A., Pal, C. J., Pascanu, R., Pezeshki, M., Raffel, C., Renshaw, D., Rocklin, M., Romero, A., Roth, M., Sadowski, P., Salvatier, J., Savard, F., Schlüter, J., Schulman, J., Schwartz, G., Serban, I. V., Serdyuk, D., Shabanian, S., Simon, É., Spieckermann, S., Subramanyam, S. R., Sygnowski, J., Tanguay, J., van Tulder, G., Turian, J. P., Urban, S., Vincent, P., Visin, F., de Vries, H., Warde-Farley, D., Webb, D. J., Willson, M., Xu, K., Xue, L., Yao, L., Zhang, S., and Zhang, Y. (2016). Theano: A python framework for fast computation of mathematical expressions. *CoRR*, abs/1605.02688.

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155.

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *Trans. Neur. Netw.*, 5(2):157–166.

Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13:281–305.

Bridle, J. S. (1990). Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In Soulié, F. F. and Hérault, J., editors, *Neurocomputing*, pages 227–236, Berlin, Heidelberg. Springer Berlin Heidelberg.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. (2015). Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274.

Collobert, R., Bengio, S., and Marithoz, J. (2002). Torch: A modular machine learning software library.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Le, Q., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., and Ng, A. (2012). Large scale distributed deep networks. volume 2, pages 1223–1231.

Elman, J. L. (1990). Finding structure in time. *COGNITIVE SCIENCE*, 14(2):179–211.

Gal, Y. and Ghahramani, Z. (2015). A Theoretically Grounded Application of Dropout in Recurrent Neural Networks. *ArXiv e-prints*.

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. 9:1735–80.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA. ACM.

Kuchaiev, O. and Ginsburg, B. (2017). Factorization tricks for LSTM networks. *CoRR*, abs/1703.10722.

Looks, M., Herreshoff, M., Hutchins, D. L., and Norvig, P. (2017). Deep learning with dynamic computation graphs.

Merity, S., Keskar, N. S., and Socher, R. (2017). Regularizing and optimizing LSTM language models. *CoRR*, abs/1708.02182.

Mikolov, T., Karafit, M., Burget, L., Cernock, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In *INTERSPEECH 2010, Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September*, pages 1045–1048.

Neubig, G., Dyer, C., Goldberg, Y., Matthews, A., Ammar, W., Anastasopoulos, A., Ballesteros, M., Chiang, D., Clothiaux, D., Cohn, T., Duh, K., Faruqui, M., Gan, C., Garrette, D., Ji, Y., Kong, L., Kuncoro, A., Kumar, G., Malaviya, C., Michel, P., Oda, Y., Richardson, M., Saphra, N., Swayamdipta, S., and Yin, P. (2017). Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*.

Pascanu, R., Mikolov, T., and Bengio, Y. (2012). Understanding the exploding gradient problem. *CoRR*, abs/1211.5063.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in pytorch.

Press, O. and Wolf, L. (2016). Using the output embedding to improve language models. *CoRR*, abs/1608.05859.

Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms.

Tokui, S., Oono, K., Hido, S., and Clayton, J. (2015). Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*.

Wan, L., Zeiler, M., Zhang, S., Cun, Y. L., and Fergus, R. (2013). Regularization of neural networks using dropconnect. In Dasgupta, S. and McAllester, D., editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1058–1066, Atlanta, Georgia, USA. PMLR.

Xu, W. and Rudnicky, A. (2000). Can artificial neural networks learn language models? In Xu and Rudnicky (2000), pages 202–205.

Yang, Z., Dai, Z., Salakhutdinov, R., and Cohen, W. W. (2018). Breaking the softmax bottleneck: A high-rank RNN language model. In *International Conference on Learning Representations*.

Zaremba, W., Sutskever, I., and Vinyals, O. (2014). Recurrent neural network regularization. *CoRR*, abs/1409.2329.