ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Improving Language Modeling with Recurrent Neural Networks

Valentin JULIA

École Polytechnique Fédérale de Lausanne
School of Computer and Communication Sciences

March 2015

Master Thesis

Supervisor
Dr. Thomas Kemp
Sony / SSG Sony

Supervisor
Prof. Hervé Bourlard
EPFL / IDIAP

Speech and Sound Group, European Technology Center
Sony Deutschland GmbH, Stuttgart, Germany

Abstract

Sony wants to develop its large vocabulary speech recognition system in order to have full control over it, and to fully understand training components/resources and deployment. Two previous Sony's students have already worked on the two distinct models that make up a speech recognition system : acoustic and language. This master thesis will focus on language modeling and will compare findings with those to be found in the work of Ivan Sliječević [1].

Ivan shows that feedforward neural networks have higher accuracy than the baseline. However, training time is quite long with a small vocabulary and it takes a considerable amount of time to process a large vocabulary task. In this Master's thesis, we present new deep neural network techniques for language modeling such as recurrent neural networks (RNN) and long short-term memory (LSTM) in order to improve language modeling. We briefly introduce language modeling and its evaluation measurements. Then we introduce basic concepts and mathematical formulas for RNN and LSTM. Last but not least, we show all our results on the one billion word benchmark database.

Contents

Abstract (English) **i**

List of figures **v**

List of tables **vii**

1 Introduction **1**

2 State of the Art **3**

3 Language Model **7**
3.1 Description 7
3.2 Evaluation 8
3.2.1 Out of Vocabulary 9

4 Recurrent Neural Networks **11**
4.1 Notation 12
4.2 Training 12
4.2.1 Backpropagation through Time 13
4.3 Maximum Entropy Model 14

5 Long Short-Term Memory **17**
5.1 Notation 18
5.2 Training 19
5.2.1 Forward pass 19
5.2.2 Backward propagation 19

6 Data and Open-Source Toolkits **21**
6.1 Data 21
6.2 Toolkits 22
6.2.1 Kaldi issues 23
6.3 sDeep 26

7 Results **29**
7.1 Baseline 29
7.2 Feedforward Neural Network 30

7.3 RNN 30
7.3.1 Multi-Threading 30
7.3.2 Maximum Entropy model 32
7.3.3 Hyperparameters 34
7.4 LSTM 41
7.4.1 Initialization 41
7.4.2 First Layer Type 41
7.4.3 Stacked LSTM 42
7.4.4 Early Stopping 43
7.4.5 Hyperparameters 43
7.5 Testing 49
7.5.1 Minimal count 3 49
7.5.2 No minimal count 51

8 Conclusion **53**

A Training derivation **55**
A.1 Usual Backpropagation 55
A.2 Backpropagation through Time 57

B Parameters **59**
B.1 RNN 59
B.2 LSTM 61
B.3 Testing 63

Bibliography **68**

List of Figures

3.1 Drawback of second evaluation method for OOV 9

4.1 A simple recurrent neural network also call the Elman network. 11
4.2 The unfolded RNN with $\tau = 3$. 14
4.3 Memory complexity of ME model: Bi-gram with 3 words and the number of connection in **B** is 3^2. 15

5.1 An LSTM unit 18

6.1 Comparison of training and validation perplexity for Mikolov and Kaldi 23
6.2 Ratio when $\beta||W||$ is larger than $\eta * \nabla$ within a given range for each updating matrix. 24
6.3 Comparison of the training and validation perplexity between Mikolov, Kaldi with no regularization, Kaldi with the regularization parameter not scaled by the learning rate, and Kaldi with the regularization parameter scaled by the learning rate 25
6.4 More visual representation of equation adding term in equation (6.2). Indeed red arrows represent $\mathbf{V}^T\mathbf{e_o}(k)$, where $k = t, t-1, t-2, \ldots$, in order to backpropagate error from layer τ to layer $\tau - 1$. This figure is taken from Mikolov's Ph.D Thesis [2]. 27

7.1 Perplexity over the number of threads for Kaldi with 2M words without a maximum entropy model. 31
7.2 Comparison of 1 and 4 threads with a random initialization and maximum entropy model. 32
7.3 Perplexity of training, validation, and test set between RNN+ME model and standard RNN. 33
7.4 Validation perplexity with 7 different starting learning rates and the number of epochs needed to achieve the lowest perplexity. 34
7.5 Comparison of 2 different learning rate decays in terms of validation perplexity and number of epochs. 35
7.6 Impact of regularization term β on the validation perplexity. 35
7.7 Class distribution for 128 million training words and 100 classes. Frequent words are in class label 0 and less frequent ones in class label 99 36
7.8 Validation perplexity with different class sizes. 36

7.9 Validation perplexity with respect to `bptt` for 1M to 128M of words. 38
7.10 Training and validation perplexity with three different numbers of hidden units. 39
7.11 This graph represents the difference between baseline (Table 7.1) and the validation perplexity of Figure 7.10(b) with respect to number of words. 40
7.12 No improvement was observed if training continues for an infinite number of epochs. 43
7.13 Validation perplexity for 1 million of word with a `fixed` varying sequence size and a starting learning rate = 0.002. 46
7.14 Comparison of the validation perplexity with and without shuffling sequences and with respect to sequence size and learning rate. 47
7.15 Batch size with default (i.e. 0.01), best (i.e. lowest perplexity) learning rate and with/without shuffling. 48
7.16 For 1 and 2 millions of training word, comparison of test perplexity for KN5, feedforward neural network, RNN and LSTM with optimal parameters. All words occurring less than 3 times are mapped to `<unk>` token. 50
7.17 Comparison of achieved perplexity on test set for KN5, two RNNs and LSTM with optimal parameters and taking into account all words in training set. . . . 51

List of Tables

3.1 Drawback of the perplexity measure 9

6.1 Validation perplexity of Mikolov and β scaled by η (respectively, red and green curves in Figure 6.3(b)). 25

7.1 Perplexity of validation set and training time [hh:mm:ss] for KN5 and KN5-modified. 29

7.2 Perplexity and training time for a feedforward neural network (FFNN). Also, KN5 is depicted because the minimal count of 3 is used. Thus, it modifies the test set by mapping unseen words to `<unk>`. 30

7.3 Comparison of 1-thread and multi-threading in terms of perplexity and training time. 31

7.4 Influence of the hash size on the training and validation perplexity. 33

7.5 Training time for RNN + ME and standard RNN. 33

7.6 Number of epochs and minutes per epoch for three different class sizes: 100, 600, and 1000. 37

7.7 Relative perplexity reduction between 100 and 200 hidden units and between 200 and 400. 39

7.8 The number of epochs to train RNN and the minutes needed to finish one epoch for 100, 200 and 400 hidden units. This symbol * means that a numerical error occurred. 40

7.9 Perplexity and training time of projection layer type for 1 millions of training tokens after 15 epochs. 42

7.10 Perplexity and training time of 2-layer LSTM with 300 units for the first one and the second varies from 0 (one LSTM layer) 400. 42

7.11 Perplexity and the number epochs needed to stop with respect to a momentum parameters. 44

7.12 Perplexity and epoch according to several starting learning rates η for 1 million of training words. 44

7.13 Validation perplexity for 1 million of training words with respect to two variables: learning rate and sequence length. Several values are in red in order to emphasize that large sequence length leads to poor results. 46

7.14 Number of epoch and minutes per epoch with various batch size. No shuffling and 1 CPU are used. 47
7.15 Perplexity and minutes per epoch for 1 million training tokens and various first layer units. The number of epochs is fixed to 15. 48
7.16 Perplexity and training time with respect to the dimension of LSTM for a small training set (1 million words). 49
7.17 Improvement on test set in percentage of LSTM over baseline, feedforward neural network and RNN with a minimal count of 3. 49
7.18 Number of epoch required to stop training and minutes per epoch for all models and with a minimal count of 3. 50
7.19 Improvement on test set in percentage of the two RNNS over baseline without a minimal count. 51
7.20 Improvement on test set in percentage of LSTM over baseline and the two RNNs without a minimal count. 52
7.21 Training time for 3 different networks and the baseline with the full vocabulary. 52

B.1 Parameters of results in Section 6.2.1 "Kaldi Issues". 59
B.2 Parameters of Section 7.3.1 "Multi-Threading" 60
B.3 Parameters of Figure 7.3 and Table 7.5. For Table 7.4, same parameters as RNN+ME except ME hash is 500M 60
B.4 Parameters for all hyperparameters for RNN measurements. 60
B.5 Parameters of Randon initialization (second column) and The first layer type (third column) 61
B.6 Parameters of 2nd LSTM layer size (second column), early stopping (third column), momentum (fourth column) and learning rate (last column) 61
B.7 Parameters of Section Sequence 7.4.5 (second to fourth column) and Section Batch 7.4.5 (fifth and sixth column) 62
B.8 Parameters of First layer size (second column) and First LSTM size (third column) 62
B.9 The second and third columns are parameters for Testing section with a minimal count of 3 (Figure 7.16, Tables 7.17,7.18). Instead the three last columns contain the parameter without minimal count (Figure 7.17 and Tables 7.20,7.21). The recurrent layer type refers to hidden layer for RNN and its size is in row with variable: 1st layer. 63

1 Introduction

Language modeling is an important component of speech recognition tasks or machine translation. The original N-Gram model has been the most popular and successful technique for the last few decades. At the beginning of the century, feedforward neural networks showed similar performance when compared to N-gram. However, the former's training time is considerably longer. New techniques using recurrence, such as recurrent neural networks (RNN) and long short-term memory, have outperformed the standard model. Again, those networks are time-consuming and training them and finding the ideal parameters is not an easy task.

The goal of this thesis is to build, and find the optimal parameters for, the optimal network in order to improve the baseline for language modeling. This means that we go through all the parameters for RNN and LSTM—such as learning rate, dimension of different layers, batch, class size for RNN, and sequence for LSTM—and try to find the optimal values. We use the "one billion word benchmark dataset" divided into multiple shards of 2^0, 2^1 up to 2^9 and 792 million training words. We needed such division because, handling such a huge training set, convergence took too long. Thus, we train our model with those shards. Of course, some preprocessing steps such as tokenization and splitting the training set into a validation set (20%) and a new training set (80%) have been performed.

Additionally Sony wants to develop its tool in order to fully understand and control it. So we updated this toolbox by adding RNN, and its training algorithm : backpropagation through time (BPTT). BPTT is an extension of normal backpropagation because propagating the error through a recurrent layer is not possible with normal backpropagation, but BPTT solves it. Implementation of RNN is difficult. Indeed, even if the model is trained and converges, errors are possible. In order to ensure that Sony's tool does not contain any errors we compare it with another toolkit to see if the final results are identical. LSTM's architecture is different from other networks because it introduces new gates that allow a unit (or neuron) to flush its internal memory or propagate an output. We did not implement it in sDeep due to time constrainsts.

Different open source toolkits are available with their advantages (multi-threading, mixing models) and their drawbacks (early stopping not implemented, a specific input file format). We compare those toolkits to find the best one for RNN and LSTM; we then look for the optimal hyperparameters for both networks. Finally, we compare our best models with baseline in order to see if recurrent neural networks (RNN and LSTM) outperform older architecture. RNN improves significantly on standard baseline (19% for 8 million training words) and has similar performance to that of feedforward neural networks (FFNN). Moreover, performance is even better with LSTM, which outperforms our best RNN; the relative perplexity reduction is about 15% for 8 million training words. However, N-Gram training time is faster than any other model—less than a minute, whereas RNN and LSTM training time is in the order of a couple of hours up to three weeks.

This Master's thesis is orgainzed as follows: The first section concerns prior work carried out with RNN and LSTM for language modeling purposes, then—in the next two sections—we briefly explain what language modeling is and how can we evaluate it. The fourth and the fifth sections are dedicated to the theoretical aspects of RNN and LSTM, respectively. The sixth section is devoted to the data and toolkits we needed to obtain results. Finally, we report all our results in order to find optimal parameters and compare RNN and LSTM with respect to the baseline.

2 State of the Art

During the last few decades, several approaches have been developed for language modeling including the N-Gram. There exist multiple variants of N-Gram ; Katz's backoff model [3] or the Kneser-Ney (KN) N-Gram [4]. In many publications, it has been reported that modified Kneser-Ney smoothing achieves the best results [5]. N-Gram is mainly used because its training speed is high and evaluation is fast. Also, the results obtained are excellent. However, N-Gram does not take into account a larger context than $N-1$ previous words or the similarity between two sentences [6]. If we observe a sentence in the training corpus, this should help us to generalize sentences that are semantically similar. Finally computational memory increases exponentially as N becomes large, and so N is usually around 5. To compare results, we will use the Kneser-Ney 5-Gram as the baseline. The difference between the modified KN 5-Gram and the original is insignificant as we report in the results section.

Language modeling based on the neural network technique was first demonstrated by Elman [7] at beginning of the 1990s. However, the attention given to neural networks faded until early 2000 when Bengio's work was published [6]. Significant results included the fact that a neural network performs better than a modified KN 5-Gram with the Brown Corpus (around 1M words for training, 200K for validation, and 100K for testing) and the Associated Press Corpus (14M words for training and 1M for validation and testing). Regarding the Brown Corpus, a neural network is 24% better; and this figure is about 8% on the Associated Press Corpus.

After Bengio's successful paper [6], several papers explored neural network based language models [8, 9, 10, 11, 12]. Emami and Jelinek [10] compare a neural network and a linear combination of neural network and N-Gram results. The averaged reduction in word error rate is 2% on the Wall Street Journal (930K words for training, 74K and 82K for, respectively, heldout and testing set). In [11] the word error rate is decreased by up to 0.4% and perplexity up to 0.9% on the French Broadcast Corpus (up to 21M words for the neural network and around 600M words for the backoff LM).

The main drawback of neural networks is the significant computational cost : numerous parameters must be trained with a massive amount of data. Unfortunately, large datasets leads to a more accurate model. All previous works (except that of [12]) use a small quantity of training data (less than 25M words). However, even with large corpora [12], the reported results are either above the baseline or very close (regarding perplexity and word error rate).

N-Gram remained the best algorithm to model languages because feedforward neural networks had no significant improvements and were computationally expensive and could take several days to train. However, in 2010, Tomas Mikolov stated that neural networks beat the baseline [13]. Instead of using the usual feedforward neural network, Mikolov introduced a simple recurrent neural network with one hidden layer and one recurrent state (it is also called Elman's network [7]). The perplexity is better by roughly 20% with RNN and 30% using a mixture of RNN and KN 5-Gram. In terms of word error rate, the reduction is about 18% with the best model. The results came from routines performed with 6.4M (about 300K sentences) words from the Wall Street Journal database.

In 2011, Mikolov tried to improve his model[13] in terms of speed and accuracy. In order to speed up the training, the words are clustered (factorization of the output layer) according to the number of times they occur in the training set. This means that if the number of classes is set to 20, the first 5% of the unigram probability distribution is mapped to the first class and so on. For example, in the Penn Treebank Corpus, the word the represents 5% of the corpus, thus the class 1 contains only the article *the*. The second speed up to training consists of adding a compression layer [14] in order to reduce computational complexity, and the total number of parameters. With both speed-up techniques, training time is lower and accuracy (regarding perplexity) is not impaired.

Neural networks were delivering better performance than KN 5-Gram but only for a small corpus (about 6M words). How about results with a large corpus (a few billion words)? We know that training time depends significantly on the number of words, but it is possible to reduce it using Mikolov's speed-up techniques [14]. To handle large corpora, Mikolov proposed some strategies [15]. In order to avoid a training time that exceeds a couple of days or a week, he reduced the number of epochs, the training tokens, the hidden units (number of neurons in the hidden layer), and the vocabulary. For the latter he set a minimal count, meaning all words that have a frequency (number of occurrences) less than the minimal count are considered to be out-of-vocabulary (OOV). Other strategies to speed up the process have been introduced by Bengio ; sampling the training token [16] and parallelization of the computations [6]. The latter has been extended by Schwenk [17] with floating point representation and batch mode.

At least Mikolov improved RNN with a maximum entropy model (ME) that can be trained as part as the whole neural network, and it can be seen similar to direct connections between the input and output layers. Additionally, training time is decreased since fewer iterations are needed. Since input and output vectors have a size of the vocabulary, representing a tri-gram with an ME model implies V^3 connections, , where V is the vocabulary size. Indeed all possible pairs (V^2) of words are connected to one possible output word (V). $V^2 * V$ connections are then needed to fully connect pairs with output. Thus, with a vocabulary of 100K, the model becomes infeasible because it requires too much memory. In order to solve this issue, a hash-based implementation is used. Of course, some collisions occur—thus the size of the hash is a crucial parameter. For an RNN with an ME model with a hash of 10^9 parameters, perplexity is about 30% better compared to an RNN with 80 hidden units, and 5% better than a KN 4-Gram baseline.

Unfortunately training RNN is hard; indeed, training with a stochastic gradient descent using backpropagation through time (Section 4.2.1) the gradient vanishes or grows exponentially. Thus, it becomes difficult to learn long-term dependencies [18]. Mikolov proposes to update the weights that connect input to a hidden layer and the weights that connect the hidden to a recurrent layer after processing at least 10 training words [2].

Another approach to improving training is called long short-term memory (LSTM) and was first introduced by Hochreiter and Schmidhuber [19]. This architecture allows a constant flow of the error and, because the gradient depends on the backpropagated error (Equations (4.4), (4.7) and (4.8)), if error flow remains constant, the gradient does not vanish. Each cell of an LSTM unit has two gates: input and output. These gates have to be trained to learn, respectively, what information has to be stored in the memory and when to read it out. Gers et al. [20] point out that this architecture has a major weakness: with continuous input sequences (that are not explicitly segmented) the LSTM unit could saturate. Resetting the internal unit value at the end of the sequence solves this weakness. Thus [20] introduced a third gate: forget, that learns how long information should be stored in the cell's memory.

LSTM has remarkably improved accuracy for speech recognition purposes [21, 22] and for language modeling [23]. In the latter, perplexity is lower by about 8% and the word error rate is significantly lower with a simple architecture: the first layer is considered as a projection layer and the second layer is the recurrent layer (LSTM). Two corpora are combined (English Treebank and Quaero (a French corpus)) with a total of 28M training tokens and 117K for testing.

The standard input vector for LSTM is 1-out-of-V (where V is the size of the vocabulary), and a projection is then performed to reduce the dimension of the input vector for the LSTM layer. In [24], latent Dirichlet allocation (LDA) or classes are used in order to replace the 1-out-of-V input vector and the projection layer. Mixing LSTM with LDA and KN5 outperforms baseline. However, no comparison is carried out between LSTM with 1-out-of-V and with LDA.

3 Language Model

Since the beginning of the 1990s, improvements made in automatic speech recognition have been considerable. Today, these system are widely used. For example, SIRI for Apple products and Cortana for Windows Phone are embedded in our mobile phones; recognizing, understanding, and answering users' questions. Understanding and answering are not part of this thesis. Instead, we will focus on the question of recognition, which is-itself- also divided into three main models: acquisition, acoustics modeling, and language modeling.

3.1 Description

Acquisition, also called feature extraction, extracts essential characteristics from the input signal. Then the recognition of those features is performed in two steps: given an input vector, it recognizes the most probable sound units (usually phonemes). On top of the acoustics models, a language model is built to find the sequence of words with the highest probability. Then, both models are combined with the most probable sequence of words. From a more mathematical point of view, the input is an utterance $\mathbf{X}$, and the system returns the most likely sequence of words $S^* = \underset{W_i}{\operatorname{argmax}} P(\mathbf{W_i}|\mathbf{X},\Theta)$ where Θ is a set of parameters and $\mathbf{W_i}$ is a word sequence. Using Bayes' rule:

$$\frac{P(\mathbf{W_i},\mathbf{X},\Theta)}{P(\mathbf{X},\Theta)} = \frac{P(\mathbf{X}|\mathbf{W_i},\Theta)P(\mathbf{W_i},\Theta)}{P(\mathbf{X}|\Theta)P(\Theta)} = \frac{P(\mathbf{X}|\mathbf{W_i},\Theta)P(\mathbf{W_i}|\Theta)}{P(\mathbf{X}|\Theta)} \tag{3.1}$$

Since $P(\mathbf{X}|\Theta)$ does not depend on $\mathbf{W_i}$, we can rewrite the original equation as

$$S^* = \underset{\mathbf{W_i}}{\operatorname{argmax}} P(\mathbf{X}|\mathbf{W_i},\Theta_A)P(\mathbf{W_i}|\Theta_L), \tag{3.2}$$

where $P(\mathbf{X}|\mathbf{W_i},\Theta_A)$ is the acoustic model and Θ_A the set of parameters associated to acoustic modeling and $P(\mathbf{W_i}|\Theta_L)$ is the language model. Acoustics modeling is beyond the scope of this Master thesis. However, one should know that acoustic modeling is solved with a statistical model (e.g., Gaussian Mixture Model, Hidden Markov model) or neural networks (deep neural networks). Additionally, it is essential for an accurate recognition system and for language modeling.

The goal of language modeling is to estimate the prior probability of a sequence of words or, more, to paraphrase; the goal is to build a model that is the closest to the distribution of a natural language. So $P(\mathbf{W_i})$ (we deliberately omit Θ_L to simplify the notation) operates at the sentence level, but it is infeasible to compute the probability because there are an infinite number of different sequences. However, working at the word level is simpler. $P(\mathbf{W_i}) = P(w_1, w_2, ..., w_T)$ where each w_i represents a word. Using the chain rule, we obtain

$$P(w_1, w_2, ..., w_T) = P(w_T|w1, ..., w_{T-1}) \cdot P(w_{T-1}|w_1, ..., w_{T-2}) \cdots P(w_1) \tag{3.3}$$

3.2 Evaluation

There exist different methods for evaluating the accuracy of language modeling. Perplexity (PPL) and word error rate (WER) are the most commonly used and both have their advantages and drawbacks. WER is computationally expensive and requires a full speech-recognition system. In comparison, the computation of perplexity is straightforward :

$$\begin{aligned} PPL &= \sqrt[K]{\prod_{k=1}^{K} \frac{1}{P(w_k|w_{k-1} \ldots w_1)}} && (3.4) \\ &= 2^{-\frac{1}{K}\sum_{k=1}^{K} log_2 P(w_k|w_1 \ldots w_{k-1})} && (3.5) \end{aligned}$$

Equation (3.5) can also be rewritten with the cross entropy function :

$$H(p,q) = -\sum_x p(x) log_2 q(x), \tag{3.6}$$

where $p(x)$ is the true distribution of words in a language and $q(x)$ is the probability distribution estimated during the training. Since the exact distribution can not be calculated, an estimate of the cross entropy can be defined:

$$\begin{aligned} q(w_k) &= p(w_k|w_{k-1} \ldots w_1) && (3.7) \\ H(\tilde{p}, q) &= -\frac{1}{N}\sum_k log_2 p(w_k|w_{k-1} \ldots w_1) && (3.8) \\ PPL &= 2^{H(\tilde{p},q)} && (3.9) \end{aligned}$$

According to Goodman [25], there is a positive correlation between the perplexity of a language and the system's performance. The main drawback of perplexity is that it hides achievement. The relative measurement of the percentage is often used to show improvements. Table 3.1, extracted from Mikolov's PhD. thesis [2], illustrates this drawback. Despite a constant relative reduction in perplexity, the relative reduction of entropy varies substantially. Nevertheless, the

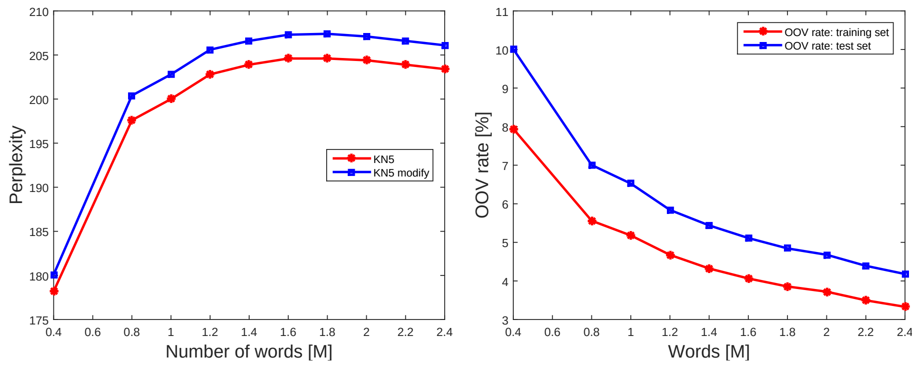
PPL	PPL after reduction	Relative PPL reduction	Entropy	Entropy after reduction	Relative entropy reduction
2	1.4	30%	1	0.49	51%
20	14	30%	4.32	3.81	11.8%
200	140	30%	7.64	7.13	6.7%
2000	1400	30%	10.97	10.45	4.7%

Table 3.1: Drawback of the perplexity measure

perplexity measurement is correct for quick comparisons, but entropy is more accurate than perplexity.

3.2.1 Out of Vocabulary

Any words not observed in the training set are considered to be out-of-vocabulary (OOV) and evaluation with OOV is performed in multiple ways: OOVs are discarded or mapped to an unknown token (`<unk>`). The first option might not be accurate "when using different tokenization strategies for morphologically complex languages"[1] however it is the simple way to evaluate OOV. The second way could be possible if the unknown token is trained during the training phase. Nevertheless with a small vocabulary the OOV rate (number of `<unk>` tokens divided by the total number of words) is too high and lead to a truncated perplexity. Indeed, the KN5-baseline for 400K words has a lower perplexity than the 1M, and 1M has a lower perplexity than 2M (cf Figure 3.1(a)). This is contradictory because the larger our training data, the better should be our language model, meaning perplexity is low. On the other hand, `<unk>` occurs too many times and becomes highly predictable, thus overall perplexity becomes low.



(a) Perplexity (b) OOV rates

Figure 3.1: Drawback of second evaluation method for OOV

We work out that when the number of training words is less than 1.6M then the OOV rate is so high that perplexity decreases, even though our language model is not improved. In order to

[1]http://www.speech.sri.com/projects/srilm/manpages/srilm-faq.7.html

take into account the OOVs' cost, the unknown word is split into letters. However, each letter should be trained during the training phase.

4 Recurrent Neural Networks

In the following chapter, we explain what a recurrent neural network (RNN) is and introduce a new training algorithm that is related to the usual backpropagation algorithm. The figure 4.1 provides a view of the difference between a simple neural network and a recurrent one. The introduction of the layer referred to as the recurrent layer allows the model to propagate previous information for an arbitrarily long time. By comparison, a deep neural network (DNN) (e.g., 6 or 7 hidden layers) uses a fixed length of context (i.e., number of hidden layers) that should be defined before the training takes place. Since we do not want to be restricted by the context length, RNN solves this restriction. Indeed at time t, $\mathbf{s_t}$ depends on $\mathbf{s_{t-1}}$ that depends on $\mathbf{s_{t-2}}$ and so one. Therefore, RNN can represent an infinite number of steps.

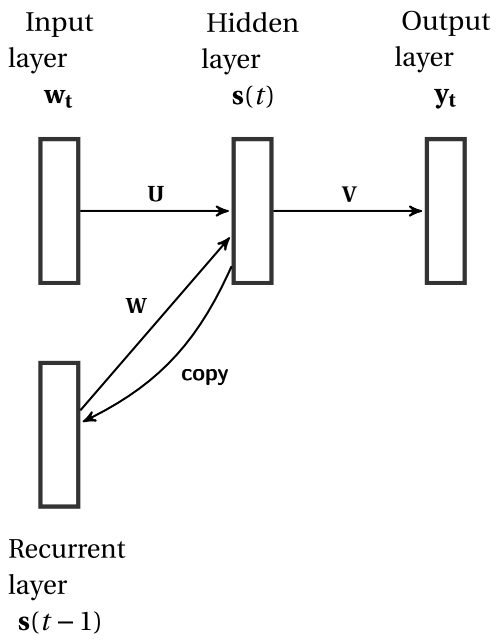

Input layer $\mathbf{w_t}$
Hidden layer $\mathbf{s}(t)$
Output layer $\mathbf{y_t}$
U
V
W
copy
Recurrent layer $\mathbf{s}(t-1)$

Figure 4.1: A simple recurrent neural network also call the Elman network.

With a DNN configuration, each layer processes a part of a task and forwards its results to the upper layer; thus introducing some hierarchy [26]. This hierarchy is the great success of DNN; simple RNN is different since it has only one hidden layer, and there is, thus, no hierarchical processing. Instead, it introduces memory. Indeed, at each epoch new information, provided by the recurrent layer, is added to the hidden layer.

The first step in the training is to define the variables and their meanings; the following section is devoted to, and the next section introduces the equations needed for training a model.

4.1 Notation

t	:	Time index
V	:	Vocabulary size
N	:	Number of words
H	:	Number of hidden units in the hidden layer
β	:	Regularization parameter
η	:	Learning rate
$\mathbf{w}_t$	:	Input vector at time t of size V x 1
$\mathbf{s}(t)$	:	Vector containing hidden units' value at time t of size H x 1
$\mathbf{s}(t-1)$	:	Vector containing recurrent units' value at time t of size H x 1
$\mathbf{y}_t$	:	Output vector at time t containing the probability of the next word of size V x 1
$\mathbf{d}_t$	:	Desired output-i.e., the expected word at time t+1. Size is V x 1
$\mathbf{e}_o(t)$	:	Error vector at the output layer of size V x 1
$\mathbf{e}_h(t)$	:	Error vector at the hidden layer of size H x 1
U	:	Weight matrix between input and hidden layer of size H x V
W	:	Weight matrix between recurrent and hidden layer of size H x H
V	:	Weight matrix between hidden and output layer of size V x H

4.2 Training

Training an RNN is performed with stochastic gradient descent, and it can be split into two parts: forward pass and backpropagation. In the following section, we write the equation needed for training. For complementary information, refer to Appendix A. The cost function (i.e., the function the neural network wants to minimize) is the cross-entropy Eq. (3.6), because it performs well for RNN in language modeling [13]. The activation function is the sigmoid Eq. (4.1). However, the tangent hyperbolic could be used instead. A softmax activation function is performed at the output layer, Eq. (4.2), to ensure the output is a probability (i.e. $0 < \mathbf{y}_t < 1$).

$$f(x) = \frac{1}{1+e^{-x}} \tag{4.1}$$

$$g(x_l) = \frac{x_l}{\sum_k e^{x_k}} \tag{4.2}$$

The forward pass is roughly the same as a one hidden layer neural network. The only difference lies in the computation of the hidden state $\mathbf{s}_t$: the value of the previous state $\mathbf{s}_{t-1}$ should be taken into account. Nevertheless the calculation of $\mathbf{s}(t)$ and $\mathbf{y}(t)$ is straightforward, we depict only the vector notation (Eq. (4.3)), a more detailed version is in appendix A.

$$\begin{aligned} s(t) &= f(\mathbf{W}\mathbf{s}(t-1)+\mathbf{U}\mathbf{w}_t) \\ y(t) &= g(\mathbf{V}\mathbf{s}(t)) \end{aligned} \tag{4.3}$$

The error at the output layer is simply the difference between the desired output and the output (i.e $\mathbf{e}_o(t) = \mathbf{d}_t - \mathbf{y}_t$). With stochastic gradient descent using the usual backpropagation, the update of **V** is the following

$$\mathbf{V} = \mathbf{V} + \eta \mathbf{s}(t)\mathbf{e}_o(t)^T + \beta\|\mathbf{V}\| \tag{4.4}$$

The error is propagated from the output to the hidden layer:

$$\mathbf{e}_h(t) = d_h\left(\mathbf{V}^T\mathbf{e}_o(t), t\right), \tag{4.5}$$

where $d_h(x,t)$ is applied element-wise :

$$d_h(\mathbf{x},t) = \mathbf{x} \bullet \mathbf{s}(t)\,(1-\mathbf{s}(t)) \tag{4.6}$$

Finally, the matrices **U** and **W** are updated as follow :

$$\mathbf{U} = \mathbf{U} + \mathbf{w}_t\mathbf{e}_h(t)^T + \beta\|\mathbf{U}\| \tag{4.7}$$

$$\mathbf{W} = \mathbf{W} + \mathbf{s}(t-1)\mathbf{e}_h(t)^T + \beta\|\mathbf{W}\| \tag{4.8}$$

4.2.1 Backpropagation through Time

The forward propagation is performed as in equation (4.3). Unfortunately, the backpropagation algorithm is somehow different. Training with the usual backpropagation algorithm is not optimal [2]. Indeed the prediction of a word depends only on the input ($\mathbf{w}_t$ eq. (4.7)) and the last hidden state value ($\mathbf{s}(t-1)$ eq (4.8)), but no information from $\mathbf{w}_{t-1}$, $\mathbf{w}_{t-2}$, $\mathbf{w}_{t-3}$ is used to update the network's matrices .

The solution is an extension of the backpropagation algorithm called backpropagation through time (BPTT). The error is backpropagated through the recurrent layer, and thus the network can remember information from previous inputs. Ideally the error is backpropagated into an infinite number of time steps. However, this is costly and backpropagation through 5 time-steps is sufficient [14]. We define the variable τ as the number time steps (we also refer to `bptt`). The new network can be seen as a deep neural network with τ hidden layers [27] and is illustrated in Figure 4.2.

The feedforward propagation remains the same as that we defined previously. However, the

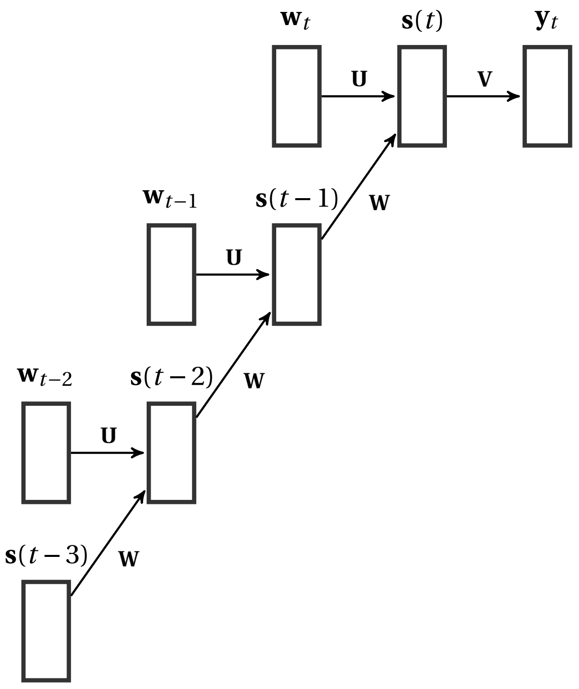


Figure 4.2: The unfolded RNN with $\tau = 3$.

propagation of errors from the hidden layer to the previous state is performed recursively:

$$\mathbf{e}_h(t-\tau) = d_h\left(\mathbf{W}^T\mathbf{e}_h(t-\tau+1), t-\tau\right), \tag{4.9}$$

where $d_h(x, t)$ is the same as Eq. (4.6). Equations (4.7) and (4.8) can be rewritten:

$$\mathbf{U} = \mathbf{U}+\eta \sum_{k=0}^{\tau} \mathbf{w}_{t-k}\mathbf{e}_h(t-k)^T + \beta\|\mathbf{U}\| \tag{4.10}$$

$$\mathbf{W} = \mathbf{W}+\eta \sum_{k=0}^{\tau} \mathbf{s}(t-k-1)\mathbf{e}_h(t-k)^T + \beta\|\mathbf{W}\| \tag{4.11}$$

Note that the updates of **U** (Eq. (4.10)) and **W** (Eq. (4.11)) should be carried out in one large update. Otherwise, if the updates are done incrementally, it can lead to instability in the training [2].

4.3 Maximum Entropy Model

The mathematical form of the maximum entropy (ME) model is the following:

$$P(w|w_{t-1}, w_{t-2}\cdots) = \frac{e^{\sum_{i=1}^{N}\lambda_i f_i(w,w_{t-1},w_{t-2},\cdots)}}{\sum_w e^{\sum_{i=1}^{N}\lambda_i f_i(w,w_{t-1},w_{t-2},\cdots)}}, \tag{4.12}$$

where f is a set of features and λ is a set of weights that must be trained. In their paper, Alumäe and Kurimo [28] show that ME models have similar performance compared to the KN4-baseline with N-gram features. An ME model could also be called direct connections since it can be viewed as a feedforward neural network without any hidden layers, thus providing direct connections from the input to the output layer. However, the memory complexity of ME models is V^N, where V is the size of the vocabulary (Figure 4.3). If V becomes large, then ME could be infeasible. Thus, Mikolov implemented a hash-based version [15]. The main drawback of hash-basing is the element of collision. Thus, the number of parameters has to be large for large vocabulary. For example, in his paper [2], with a hidden layer of 80 units, Mikolov [2] achieved the lowest perplexity with 10^9 parameters. We did an experiment with 512M words with RNN and hash-based ME, but we had a memory issue, thus even for a large vocabulary hash-based ME could not work.

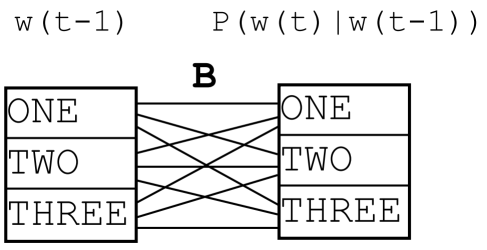


Figure 4.3: Memory complexity of ME model: Bi-gram with 3 words and the number of connection in **B** is 3^2.

5 Long Short-Term Memory

First introduced by Hochreiter and Schmidhuber[19], LSTM suffered from several weaknesses that were solved by Gers et al. [20]. We call LSTM the architecture described in [20], meaning that an LSTM unit contains three gates—input, output, and forget—and one or multiple memory cells (Figure 5.1 taken from [23]). Each memory cell has a unit called a constant error carousel (CEC) that solves the vanishing error issue, because, in the absence of new inputs or errors, the internal error value remains constant. The input (output) gate is a linear combination of the input (output) of another memory cell. The input and output gates should be trained to protect the current cell from irrelevant inputs and other units from irrelevant content in the memory. Gates are closed if activation is around zero.

The forget gate was introduced by Gers [20]; it learns when to reset the memory content once it is irrelevant. These three gates are multiplicative, meaning that each white circle is the multiplication of the input arrows. Later LSTM has been improved with a peephole connection [29]. Now it allows all the gates to have the value of the current state even if the output and input gates are closed. This peephole connection is represented by the three arrows leading from the gray circle to the blue circles.

In feedforward neural networks the value of b_i is a function of the input a_i but in an LSTM unit b_i depends on the forget, input, and output gate and three factors b_ι, b_ω and b_ϕ (three white circles), where each of them $\in (0,1)$. In order to satisfy the last property, a sigmoid function is applied and then b_ι, b_ω and b_ϕ are set.

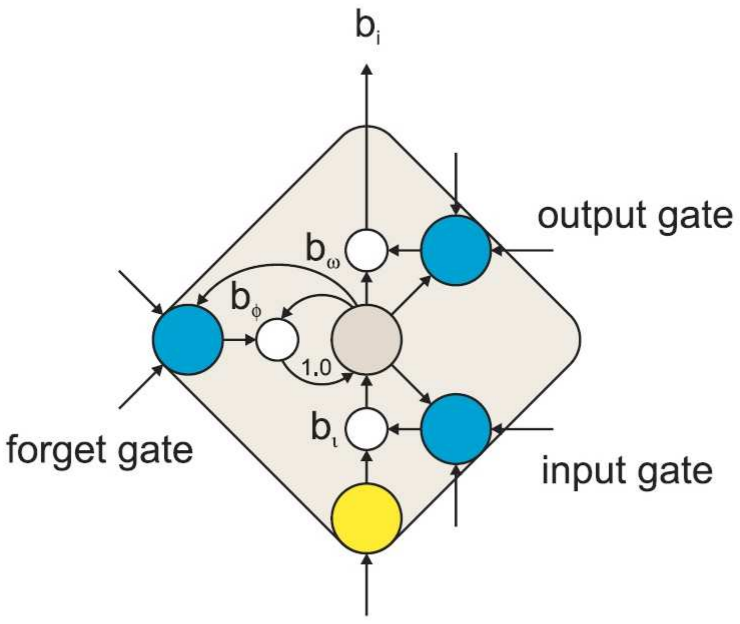


Figure 5.1: An LSTM unit

5.1 Notation

S	:	Input sequence that runs from t_0 to T
η	:	Learning rate
μ	:	momentum parameter
$x_k(t)$	:	Network input to unit k at time t
$y_k(t)$	:	Activation of $x_k(t)$
$E(t)$	:	Output error at time t
$E(S)$	:	Total error of the sequence S (i.e., $\sum_{t=t_0}^{T} E(t)$)
$t_k(t)$	:	Target at time t
$\mathbf{d}_t$	:	desired output (i.e $y_k(t) - t_k(t)$)
N	:	Total number of network units (bias and input included but not input, forget, and output gates)
$\mathbf{W}_{ij}$	:	Weight from unit j to unit i
Subscript ι, ϕ, τ, k	:	Refer to input, forget, and output gate and output layer unit respectively
s_c	:	State value of cell c
f, g, h	:	activation function of gates, cell input and cell output, respectively

5.2 Training

Even though the equations are similar to RNN, the gates introduce new equations. The training is carried out with a stochastic gradient descent using a backpropagation algorithm. All equations are taken from Graves et al.'s paper [30]. For further explanation and details, please refer to Gers et al. [29] and Gers [31].

5.2.1 Forward pass

Like RNN the forward pass is trivial and for each LSTM unit, does the following:

$$\begin{aligned} x_\iota(t) &= \sum_{j\in N} w_{\iota j} y_j(t-1) + \sum_{c\in C} w_{\iota c} s_c(t-1) && (5.1)\\ y_\iota(t) &= f(x_\iota(t)) && (5.2)\\ x_\phi(t) &= \sum_{j\in N} w_{\phi j} y_j(t-1) + \sum_{c\in C} w_{\phi c} s_c(t-1) && (5.3)\\ y_\phi(t) &= f\left(x_\phi(t)\right) && (5.4) \end{aligned}$$

For each cells:

$$\begin{aligned} x_c(t) &= \sum_{j\in N} w_{cj} y_l(t-1) && (5.5)\\ s_c(t) &= y_\phi(t) s_c(t-1) + y_\iota(t) g(x_c(t)) && (5.6) \end{aligned}$$

Finally, for output gates and cell outputs:

$$\begin{aligned} x_\omega(t) &= \sum_{j\in N} w_{\omega j} y_j(t-1) + \sum_{c\in C} w_{\omega c} s_c(t) && (5.7)\\ y_\omega(t) &= f(x_\omega(t)) && (5.8)\\ y_c(t) &= y_\omega(t) h(s_c(t)) && (5.9) \end{aligned}$$

5.2.2 Backward propagation

The cross-entropy criterion is used to minimize the error and with a softmax ouptut layer as for RNN. Let us define δs and for all output cells c, ϵ_c

$$\begin{aligned} \delta_\theta(t) &= \frac{\partial E(t)}{\partial x_\theta(t)} \quad \theta = \iota, \phi, \omega \text{ or } k && (5.10)\\ \epsilon_c(t) &= \sum_{j\in N} w_{jc} \delta_j(t+1) && (5.11) \end{aligned}$$

Output gates:

$$\delta_\omega = f'(x_\omega(t)) \sum_{c\in C} \epsilon_c h(s_c(t)) \quad (5.12)$$

For the internal state (as for RNN this is done recursively):

$$\begin{aligned}\frac{\partial E(t)}{\partial s_c(t)} &= \epsilon_c y_\omega h'(s_c(t)) + \frac{\partial E(t+1)}{\partial s_c(t+1)} y_\phi(t+1) + \\ &\quad \delta_\iota(t+1)\, w_{\iota c} + \delta_\phi(t+1)\, w_{\phi c} + \delta_\omega w_{\omega c} \qquad (5.13)\end{aligned}$$

For internal cells, and forget and input gates:

$$\delta_c = y_\iota(t)\, g'(x_c(t)) \frac{\partial E(t)}{\partial s_c(t)} \qquad (5.14)$$

$$\delta_\phi = f'\left(x_\phi(t)\right) \sum_{c\in C} \frac{\partial E(t)}{\partial s_c(t)} s_c(t-1) \qquad (5.15)$$

$$\delta_\iota = f'(x_\iota(t)) \sum_{c\in C} \frac{\partial E(t)}{\partial s_c(t)} g(x_c(t)) \qquad (5.16)$$

Updating weights is done after processing a whole sequence S.

$$\nabla_{\mathbf{ij}} S = \frac{\partial E(S)}{\partial w_{ij}} = \sum_{t=t_0}^{T} \delta_i(t)\, y_j(t-1) \qquad (5.17)$$

$$\Delta \mathbf{W}_S = \mathbf{W}_{S+1} - \mathbf{W}_S \qquad (5.18)$$

$$\Delta w_{ij}(S) = -\eta \nabla_{\mathbf{ij}} S + \mu \Delta w_{ij}(S-1) \qquad (5.19)$$

6 Data and Open-Source Toolkits

6.1 Data

The One billion Word Benchmark [32] is used for language modeling and is available at https://code.google.com/p/1-billion-word-language-modeling-benchmark/. All scripts needed to preprocess (tokenization, remove duplicates) the text are provided. The following steps were performed to obtain the final version of the data:

- All files were selected
- A normalization (e.g. " and « should be the same) and tokenization was performed
- Duplicate sentences were removed
- OOV words were mapped to the `<unk>` token
- Sentences were randomized
- The whole dataset was split into 100 disjoint set
- 1% of data was used as a test set
- This test set was randomized

The test set contains about 8 million words, with the end sentence marker `</s>`. Up to 99 partitions could be used for language modeling, but each partition contains on average 8.3M words; thanks to Ivan Slijepčević, who wrote the script, we will perform the following text in order to have our final corpus:

- Select 1 of the 99 sets
- Randomize sentences
- Split the set into a text file of 1M, 2M, 4M and 8M words

- Construct corpora of 16M, 32M ... 512M, 800M words with the 98 remaining partitions
- For each corpus, split it into 20% for the validation set and 80% for the training set

Even though, the number of words is not 1 million in the training set, because it only 80% of the corpora of 1 million. We refer this training set as 1M or 1 million. This reference is identical to all other training sets.

6.2 Toolkits

In this section we present all open-source toolkits used to obtain all results.

SRILM [33] SRILM is a set of C++ libraries for statistical language modeling. SRILM, exclusively, was used for creating our KN5-gram baseline.

Mikolov [34] RNN for language modeling toolkit written by Tomas Mikolov. He wrote it in C++, and although the code is not easily understandable, the manipulation is straightforward. The main drawback is the absence of parallelization of computations; it can only be computed on 1 CPU (GPU not possible). However, all improvements such as the maximum entropy model, compression layer, and so on could be used with this toolkit.

Kaldi [35] RNN for language modeling is also integrated into the speech recognition toolkit Kaldi, and although the version is similar to Mikolov's, there are some differences. For example, integration of classes and compression layer are not supported by Kaldi but it does allow multi-processing.

RWTHLM [36] Rheinisch-Westfällische Technische Hochschule Language Modeling is a recurrent and LSTM toolkit written by Martin Sundermeyer. It is specific to language modeling, easy to use, has many options and, as does Kaldi, it allows multi-processing (thanks to OpenMP). However, no early stopping is performed, meaning that training ends when it reaches a maximum number of epochs or is stopped manually. Moreover, another drawback is the setting up process; we spent a couple of days to compile and build.

Currennt [37] The main advantage of the CUDA RecuREnt Neural Network Toolkit is it supports computations on GPU and/or CPU. Creating a network and giving the options is easy. However, the input data must be in a NetCDF format, and it is not suitable for language modeling as we explain in the next paragraph.

NetCDF Network Common Data Form is a self-describing and portable file format. Creating a NetCDF file requires two steps: create a text file containing all variables, dimensions, and data; transform this text file into NetCDF format. Writing data explicitly is constraining for language modeling, since the input data is sparse. Indeed, the input word has a size of vocabulary and contains only one value equal to 1 (1-out-of-vocabulary coding). Also, each input has a target of the same size. As an example: a file contains 916 words (30 sentences), and each of them is in a vocabulary of size 54,284. The size of the file is 360 MB. For this reason; we tried to find an another word representation like word2vec.

Word2vec [38] This tool, written by Mikolov et al., produces "efficient implementation of the continuous bag-of-words (CBOW) and skip-gram"[1]" in order to have a vector representation of a word. CBOW architecture works as follows: 4 previous words and 4 next words are used to predict the current word. Skip-gram predicts previous and next words given the current word [38].

6.2.1 Kaldi issues

Kaldi is designed to perform speech recognition, and it includes RNN for language modeling. First we did some comparisons with Mikolov's toolkit and, alas, the results were not alike. Of course, the initialization of the RNN and the evaluation of the model were identical. The results are depicted in Figure 6.1. The behavior of Kaldi's curves is similar to those of Mikolov; validation perplexity behaves like an exponential decrease while the training curve is oscillating. However, the Mikolov toolkit outperforms Kaldi; relative validation perplexity reduction is about 11% better for 1M words and reaches its maximum (19%) for 64 and 128M words.

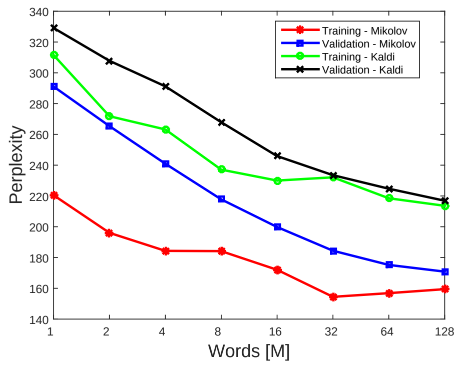


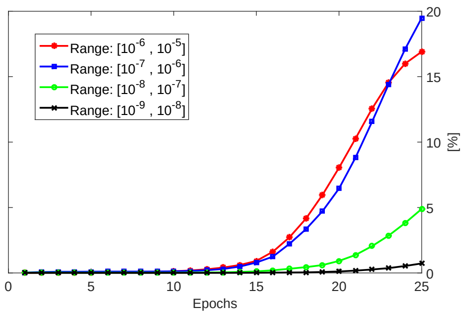
Figure 6.1: Comparison of training and validation perplexity for Mikolov and Kaldi

Updating a matrix with a regularization is done as follow:

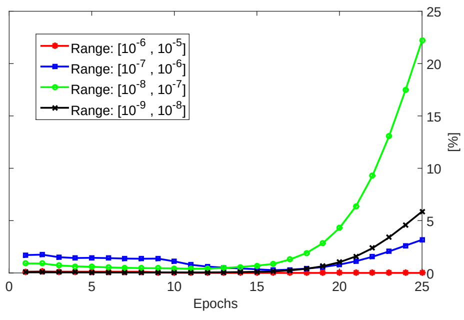
$$\mathbf{M} = \mathbf{M} + \eta\nabla + \beta||\mathbf{M}||, \tag{6.1}$$

[1]https://code.google.com/p/word2vec/

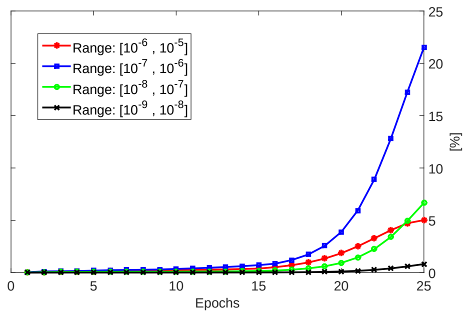
where **M** is a matrix. A significant difference exists between the two toolkits. In Mikolov, the regularization parameter β depends on learning rate: meaning that the update is done according to $\eta\left(\nabla + \beta||\mathbf{M}||\right)$. Whereas in Kaldi, the update follows Equation (6.1). When the learning rate η gets close to zero, then the learning term (i.e., $\eta\nabla$) becomes smaller than the regularization term (i.e., $\beta||\mathbf{M}||$). This means that the matrix weights are pushed up at each epoch by the regularization term, and thus that the overall performance is worse. We wanted to find at which value the learning term is smaller than the regularization term. So we computed the difference between these two terms and whenever the difference lay within a particular range we increased a counter by 1. In Figure 6.2 we display the percentage of time the difference is between 4 ranges for each epoch and for the three matrices (**U**, **W** and **V**).



(a) **U**: from input to hidden layer



(b) **W**: from recurrent to hidden layer

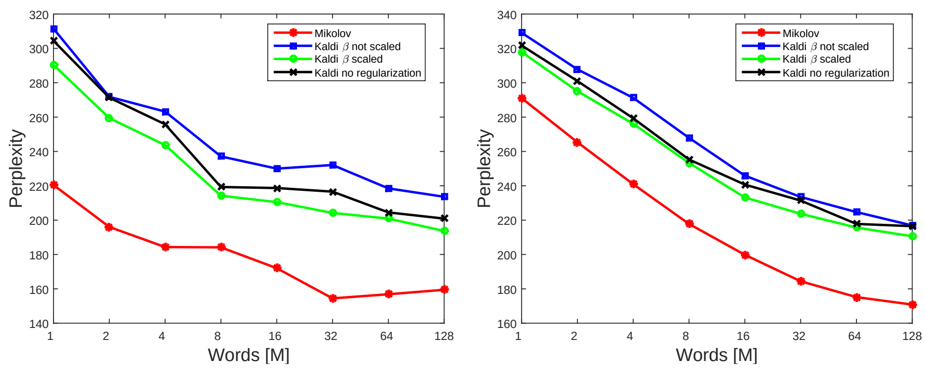


(c) **V**: from hidden to output layer

Figure 6.2: Ratio when $\beta||W||$ is larger than $\eta * \nabla$ within a given range for each updating matrix.

We select those four ranges because above 10^{-9} it becomes insignificant and below 10^{-4} the percentages are close to 0 for almost all matrices. Before epoch 15 the ratios are approximately equal to 0; this makes sense since, before epoch 15, the learning rate is around 10^{-4} whereas β is around 10^{-6}. Thus, the regularization term does not push up the weights. After epoch 15, all ratios increase, and are more important for the matrix **U**. Indeed, more than 40% of the time the regularization term is bigger than learning term for **U**. For **W**, the difference is mainly between 10^{-8} and 10^{-7}, meaning that each weight roughly increases by 10^{-7}, which is reasonable. So the fact that β is independent of η mostly affects the matrix between the input and the hidden layer.

To avoid this noisy behavior, we simply scaled the factor β by η in Kaldi. Also, we trained the network without any regularization. Our starting learning rate was 0.1 and, in order to be consistent, the value of β_{not_scaled} is equal to $\eta * \beta_{scaled}$ (i.e 0.1*10^{-6}). Scaling the regularization parameter improves a little the training and validation perplexity as is shown in Figure 6.3. Indeed Kaldi with β_{scaled} is below Kaldi with $\beta_{notScaled}$; thus, improving the validation perplexity is possible if the regularization term depends on the learning rate. We also observe that regularization does not provide outstanding improvements since performance is similar to Kaldi with β_{scaled} and without any regularization for both training and validation perplexity. Mikolov also reported that regularization does not significantly improve the language model [13].



(a) Training perplexity (b) Validation perplexity

Figure 6.3: Comparison of the training and validation perplexity between Mikolov, Kaldi with no regularization, Kaldi with the regularization parameter not scaled by the learning rate, and Kaldi with the regularization parameter scaled by the learning rate

Millions of words	Perplexity Mikolov	Perplexity Kaldi β scaled	relative PPL reduction [%]
1	291.0	317.9	8.4
2	265.4	295.1	10.1
4	240.9	276.1	12.7
8	217.8	253.3	14.0
16	199.8	233.2	14.3
32	184.4	223.7	17.6
64	175.2	215.7	18.8
128	170.8	210.6	18.9

Table 6.1: Validation perplexity of Mikolov and β scaled by η (respectively, red and green curves in Figure 6.3(b)).

Even with this dependency between β and η, Kaldi's final perplexity is far from Mikolov's as is depicted in Table 6.1. For a large vocabulary, Mikolov's toolkit performs about 20% better than Kaldi with our modification. Although the curves' behavior is the same for Kaldi and Mikolov, we suspect a bug in the Kaldi code. Therefore, all RNN experiments will be done with

Mikolov's toolkit except for multi-threading since only Kaldi is able to do this.

6.3 sDeep

Sony wants to develop its deep learning tools in order to have full control over them and understand them. It created sDeep (*s* stands for Sony and *Deep* for deep learning), which works with GPUs and it is built in Matlab. There are three different layer-types: flat, convolution, and pooling. With those layers, we can create a network and try different activation functions, cost functions, and optimizers. Also, some provided scripts are available making it possible to perform principal component analysis (PCA), auto-encoder, and independent subspace analysis (ISA).

We introduced a new layer-type: recurrent, to allow sDeep to learn and to build a recurrent neural network. The modifications for the forward pass were minor; in contrast, a new function was required to backpropagate errors through the recurrent layer. A new function was created, called `myBackPropThroughTime`, the equations for which can be found in Section 4.2.1. Further, a new internal function (`isdOptBptt`) is added to the `sdOptimize.m` file.

Some practical changes have also been carried out following the advice on training expressed by Mikolov (Section 3.3.2 in [2]). Incrementally updating the weights $\mathbf{W}$, and $\mathbf{U}$, can contribute to instability [39]. Thus, 10 to 20 words are first processed and then backpropagate the error through the network. So a new structure, called `hist`, is needed to store information about all processed words. It contains previous inputs $\mathbf{w}_{t-1}$, $\mathbf{w}_{t-2}$ ..., the previous state values $\mathbf{s}_{t-1}$,$\mathbf{s}_{t-2}$ ..., and the error backpropagated from output to the hidden layer for every time step (i.e $\mathbf{V}^T\mathbf{e_o}(t)$, $\mathbf{V}^T\mathbf{e_o}(t-1), \ldots$). The mini-batch size is defined in `bpttBlock`, and its values should be between 10 and 20 [2]. The only difference between the theoretical equations (in Section 4.2) is in Equation (4.9) or, in the form of a more visual perception, in figure 6.4:

$$\mathbf{e}_h(t-\tau) = d_h\left(\mathbf{W}^T\left(\mathbf{e}_h(t-\tau+1) + \mathbf{V}^T\mathbf{e}_o(t-\tau+1)\right), t-\tau\right) \tag{6.2}$$

With the backpropagation through time algorithm, the updates are done incrementally and depend on τ previous inputs and errors. However, with this practical implementation, the updates are done after processed `bpttBlock` inputs and in practice `bpttBlock` is bigger than τ thus the effect implied by τ is removed [2].

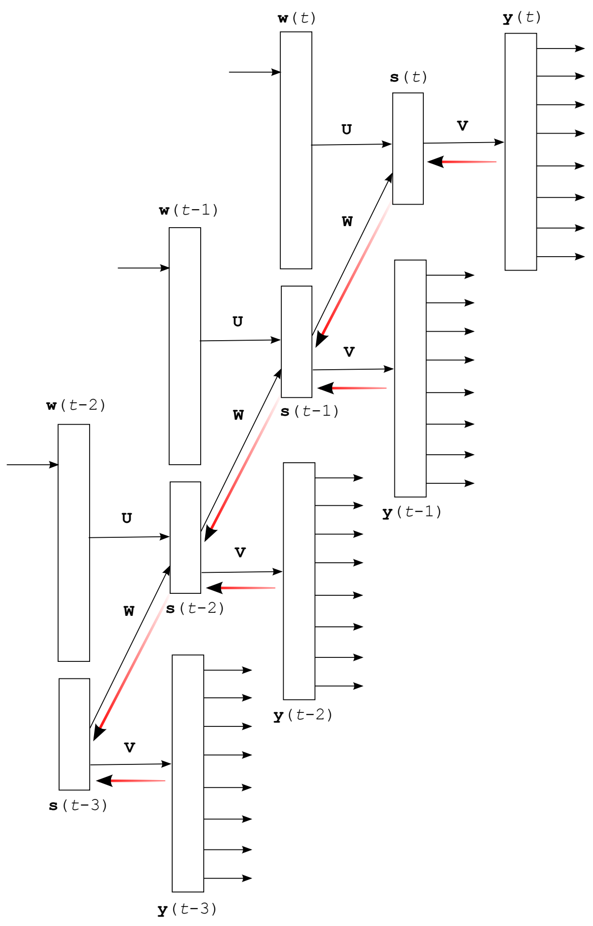


Figure 6.4: More visual representation of equation adding term in equation (6.2). Indeed red arrows represent $\mathbf{V}^T \mathbf{e_o}(k)$, where $k = t, t-1, t-2, \ldots$, in order to backpropagate error from layer τ to layer $\tau - 1$. This figure is taken from Mikolov's Ph.D Thesis [2].

7 Results

This section contains all results for RNNs and LSTMs. We did not explicitly write all parameters for every measurement in this chapter. However, they can be found in Appendix B.

7.1 Baseline

To compare our results, we first establish our baseline: KN5-gram. We compare two models: the original KN5-gram (referred to as KN5) and the modified KN5-gram smoothing [4] (called KN5-modified). Below, the comparison for both models is depicted in Table 7.1. There is a minor difference between the two models regardless of the number of words. Therefore, we choose the KN5 as the baseline, even though the KN5-modified improves the perplexity (by about 1%) slightly.

Words, in	Perplexity		Training time	
millions	KN5	KN5-modified	KN5	KN5-modified
1	294.6	292.9	00:00:10	00:00:11
2	278.5	276.3	00:00:20	00:00:23
4	254.6	252.2	00:00:37	00:00:35
8	227.9	225.4	00:01:10	00:01:06
16	201.3	198.8	00:02:08	00:02:12
32	175.9	173.6	00:04:12	00:04:20
64	152.8	150.7	00:08:21	00:10:40
128	131.6	129.8	00:20:43	00:21:30
256	111.7	110.8	00:39:51	00:49:51
512	93.0	91.9	01:29:20	01:29:53
792	81.9	81.1	02:08:11	01:58:17

Table 7.1: Perplexity of validation set and training time [hh:mm:ss] for KN5 and KN5-modified.

N-gram is widely used because the achieved perplexity is good, but also training is computationally cheap. If we look at the two last columns of Table 7.1, the training time is short, even for a large vocabulary. It is impossible to achieve such a training time with neural networks, as we will see later.

7.2 Feedforward Neural Network

To be complete regarding neural networks, we want to compare RNN and LSTM with previous work and thus include a feedforward neural network. Results are taken from Ivan's Master's thesis [1]. The parameters are the following: inputs are 4 preceding words and word at time t with each 30 dimensions, 50 hidden neurons and a starting learning rate of 2, a momentum term of 0.3, and a maximum of epochs set to 20. Two CPUs and 32 mini-batches are used and words occurring less than 3 times are mapped to the `<unk>` token in order to speed up the training. Lastly an *L2* regularization with β equals 10^{-5} is applied.

Words, in	Test perplexity		Training time [hh:mm:ss]	
millions	FFNN	KN5-modified	FFNN	KN5-modified
1	175.3	200.0	11:19:00	0:00:19
2	196.7	204.4	28:02:00	0:00:22

Table 7.2: Perplexity and training time for a feedforward neural network (FFNN). Also, KN5 is depicted because the minimal count of 3 is used. Thus, it modifies the test set by mapping unseen words to `<unk>`.

With FFNN, we observe a behavior identical to that of KN5 when handling a small vocabulary with a minimal count of 3 (Section 3.2.1). Also FFNN outperforms the usual Kneser–Ney 5-Gram, as it is shown in Table 7.2, by 12.4% for 1 million and 3.8% for 2 million training data. Unfortunately, we do not have results without a minimal count, so to compare FFNN with RNN and LSTM, we will apply a minimal count of 3 on the training set.

7.3 RNN

7.3.1 Multi-Threading

In section 6.2, we pointed out a significant advantage of Kaldi: multi-threading. A number of threads T is defined and Kaldi divides the training data into T disjoint slices for every epoch, then each slice is trained independently. Finally, the updates are done according to the last finished thread. Thus, only $\frac{1}{T}$ of the training data are taken into account to update the weights. This is unlikely since the training should update the matrices with all data to have the most accurate model. Even with an identical initialization, the final perplexity can vary with 1 or 16 threads (Figure 7.1). Indeed, all data is considered with 1 thread at each epoch and for each update, whereas only $\frac{1}{16}$ of data is used during the training for 16 threads. Moreover, updates depend on a random slice at each epoch, since the last finished thread is random.

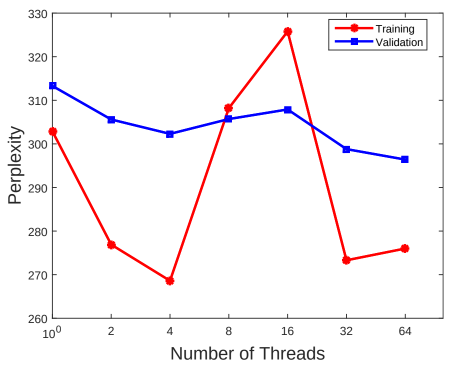


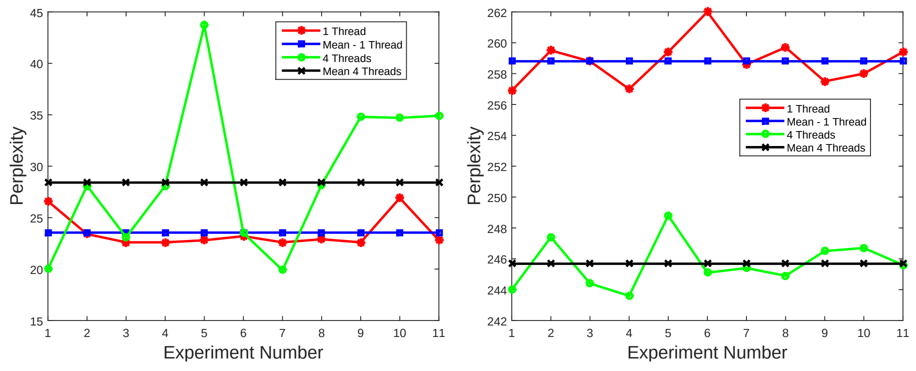
Figure 7.1: Perplexity over the number of threads for Kaldi with 2M words without a maximum entropy model.

In Table 7.3, the multi-threading has a clearly lower training time, as we expected, 21 times slower for 256 million words. It gets worse if we compare training time for KN5 (Table 7.1) with 1 thread and multi-threading: 1 thread is 304 times slower than baseline and multi-threading about 14 times (for 256M words). But we already know that neural networks can not be faster than N-Gram. Regarding the perplexity, the 1-thread result is a bit better than multi-threading for a small task and a large task, and is roughly the same for average work (between 8M and 64M) .

Words, in millions	Perplexity		Training time	
	1 Thread	10 Threads	1 Thread	10 Threads
1	280.5	300	01:03:03	00:06:38
2	261.7	260.6	03:24:50	00:07:29
4	245.3	240	07:55:17	00:16:14
8	223.1	216.2	10:52:37	00:18:46
16	205.9	197.8	22:24:56	00:32:00
32	180.6	178.8	41:31:14	01:08:43
64	164.3	169.7	62:19:49	02:39:29
128	146.3	152.5	145:55:04	05:55:11
256	133.2	142.9	202:49:54	09:20:51

Table 7.3: Comparison of 1-thread and multi-threading in terms of perplexity and training time.

We also carried out a comparison between 1 thread and 4 threads with a random initialization and the maximum entropy model in Figure 7.2. Firstly, we notice that the training perplexity has a higher variance with multiple threads than with 1 (Figure 7.2(a)). However, on the validation set, the final perplexity is much better with 4 threads than with 1 thread (Figure 7.2(b); about 5% better). In the end, the final validation perplexity is better (with or without an ME model) with 4 threads than without multi-threading. However, we can not ensure that multi-threading is always better, for example in Table 7.3, 1 thread is better than 10 threads for



(a) Training (b) Validation

Figure 7.2: Comparison of 1 and 4 threads with a random initialization and maximum entropy model.

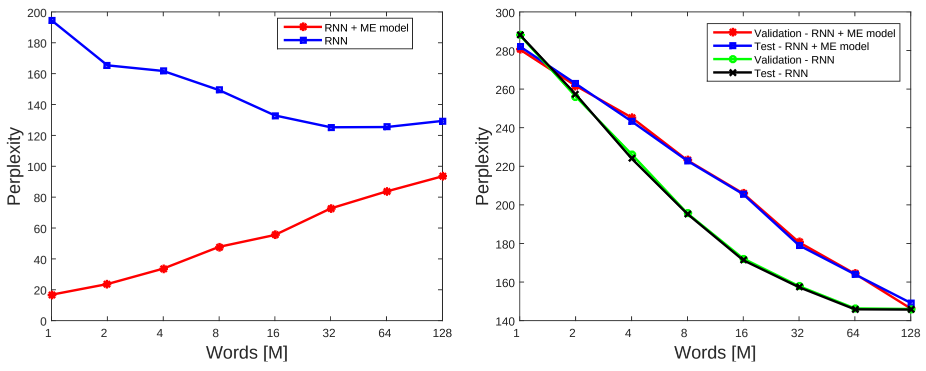
64M up to 256M. Also, it introduces some randomness with the ME model, especially for the training perplexity; for that reason, we prefer to avoid using multi-threading.

7.3.2 Maximum Entropy model

Recall that the ME model is similar to direct connections between input and output layer and it can be combined with RNN. As we explained in Section 4.3 we are not going to use the ME model for further measurements because it requires much memory even with Mikolov's hash-based implementation. However, it is interesting to compare the final perplexity with RNN and baseline. In Figure 7.3, we used a hash of 10^7 parameters with 200 hidden units. The training perplexity is shown in Figure 7.3(a), and we notice that training RNN+ME achieves better results than standard RNN. Indeed, ME can be seen as an another parallel network, and thus combining RNN+ME is comparable to mixing two networks. However, it confirms the main drawback of the ME model: once the number of words increases, there are more collisions in the hash function, thus the training perplexity increases instead of decreasing while the amount of data increases. On the other hand, the RNN's training perplexity behaves as expected; it decreases if the training set gets large.

Considering the validation and the test perplexity (Figure 7.3(b)). We achieve better results with standard RNN than with RNN+ME; this is a little surprising since Mikolov stated that RNN+ME was almost as good as RNN. Nevertheless, with a larger hash size (maybe 10^9 or larger) the final perplexity is similar. Indeed, with a hash of $5 * 10^8$, the training perplexity is lower than 10^7 (Table 7.4) because collisions are less likely to occur. Also, the validation perplexity is lower and gets closer to that of the standard RNN.

Training for RNN+ME is faster than for RNN, except for 4M words, as it is shown in Table 7.5. During all the measurements in this chapter, the data processor was shared between multiple users, and all CPUs were utilized by each user. Therefore some jobs had to be stopped and



(a) Training perplexity (b) Validation and test perplexity

Figure 7.3: Perplexity of training, validation, and test set between RNN+ME model and standard RNN.

Words, in millions	Training perplexity			Validation perplexity		
	RNN-ME: 10^7	RNN-ME: $5*10^8$	RNN	RNN-ME: 10^7	RNN-ME: $5*10^8$	RNN
1	16.8	3.4	194.5	280.5	280.0	287.9
2	23.7	22.6	165.4	261.7	254.6	256.1
8	47.8	27.7	149.3	223.1	218.8	195.6

Table 7.4: Influence of the hash size on the training and validation perplexity.

relaunched after a while. This could explain why an RNN+ME is longer than an RNN with 4M.

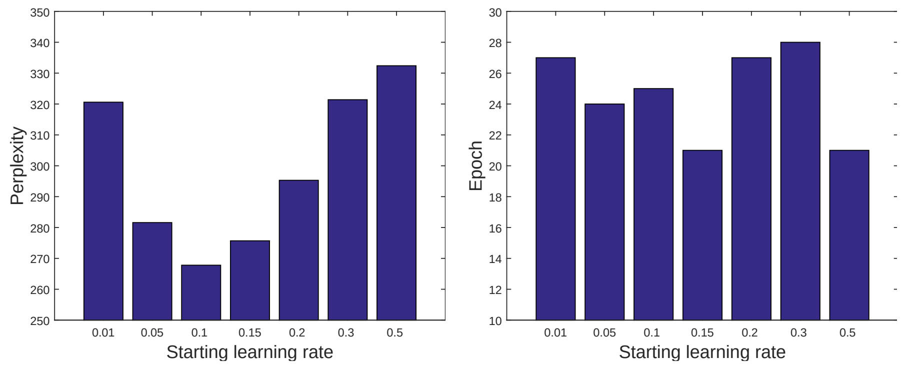
Words, in millions	Training time	
	RNN + ME	RNN
1	01:03:03	2:06:38
2	03:24:50	3:57:15
4	07:55:17	5:51:18
8	10:52:37	13:40:32
16	22:24:56	27:43:28
32	41:31:14	54:55:48
64	62:19:49	132:03:29
128	145:55:04	298:37:38

Table 7.5: Training time for RNN + ME and standard RNN.

7.3.3 Hyperparameters

Learning Rate

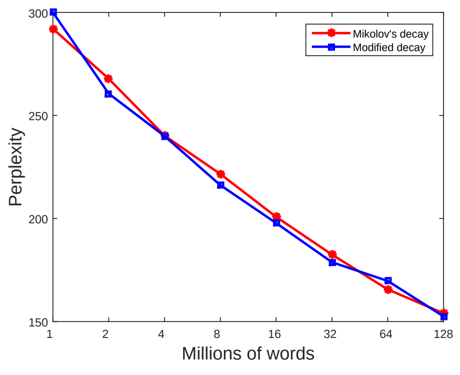
The role of the learning rate η is important; this drastically reduces the number of epochs. Therefore the convergence pace is fast with a good learning rate. On the other hand, a poor starting learning rate leads to a slow convergence pace. With a starting learning rate of 0.1, we achieve the lowest perplexity (Figure 7.4(a)) in a reasonable number of epochs (Figure 7.4(b)). In terms of speed, a learning rate of 0.15 achieves similar perplexity, but with 5 epochs less. On the other hand, a smaller learning rate (less or equal to 0.05) leads to a poor result for both perplexity and training time. We will use 0.1, despite the faster convergence with a learning rate of 0.15, because the final validation perplexity is better with 0.01. Also we can not ensure that 0.15 is always faster than 0.1, for example the number of epochs is larger with a learning rate of 0.2 than 0.1



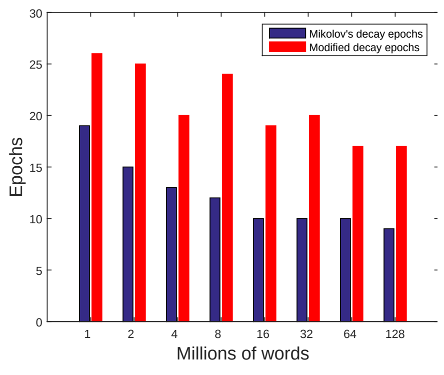
(a) Starting learning rate (b) Number of epochs

Figure 7.4: Validation perplexity with 7 different starting learning rates and the number of epochs needed to achieve the lowest perplexity.

To observe the influence of learning rate, we tried two different decays. Mikolov proposes the first one [34], which we refer to as *Mikolov's decay*. The idea is that learning remains constant until the validation entropy increases or no minimum improvement is made. Then the learning rate is halved at every epoch. Our modified decay is quite similar to that of Mikolov. The first step is the same; once no improvement is made or an increasing validation perplexity occurs, we halve the learning rate. But it remains constant for only few epochs until the validation perplexity increases or no improvements have been observed again. Our strategy converges slowly compared to Mikolov's decay. However, we expect that the final perplexity will be better. In Figure 7.5, we see that the two decay strategies reach the same perplexity on the validation set. Regarding the number of epochs (Figure 7.5(b)), the difference is big: it took (on average) 9 more epochs with our modified decay strategy. So both approaches lead to the same perplexity, but with significantly fewer epochs with Mikolovis decay. Thus, Mikolov's decay will be used for the following experiments.



(a) Perplexity



(b) Epochs

Figure 7.5: Comparison of 2 different learning rate decays in terms of validation perplexity and number of epochs.

Regularization

Regularization should prevent any excessive increase in weight in order to avoid overfitting. The *L1* regularization is used in all experiments: for all matrices, a regularization term $\beta||\mathbf{W}||$ is added. However, the regularization is applied to every 10 processed words, no matter the time step value and β is scaled by the learning rate η (Section 6.2.1).

When training small amounts of training data, regularization is needed. Indeed, overfitting is most likely to occur with a small dataset, and-since regularization is such as to avoid this-we obtain better results with β equals 10^{-4}. Also regularization must be strong enough for a small training set because the validation perplexity is significantly lower with 10^{-4} than with 10^{-7}. However, when a training set is large enough, the gap between 10^{-4} and 10^{-7} decreases like in Figure 7.6, because the training data is sufficient and so overfitting disappears. Indeed, with 32 million words, a high β penalizes the perplexity, but regularization remains useful since β equal to 10^{-6} performs better than 10^{-7} or smaller. For example, with 128M, a regularization of 10^{-6} makes an improvement of 2.8% compared to a regularization of 10^{-7}.

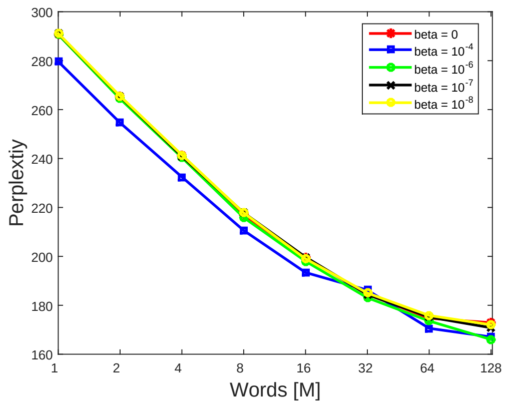


Figure 7.6: Impact of regularization term β on the validation perplexity.

Class

Classes are used in order to reduce the dimension of the output layer ("factorization of the output layer" [14]). The idea is to gather less frequent words within the same class while most frequent ones are in a separate class. Instead of learning the probability of a word, the network estimates the class probability and then the probability of words within the class [14]. Figure 7.7 depicts the number of words within all class labels.

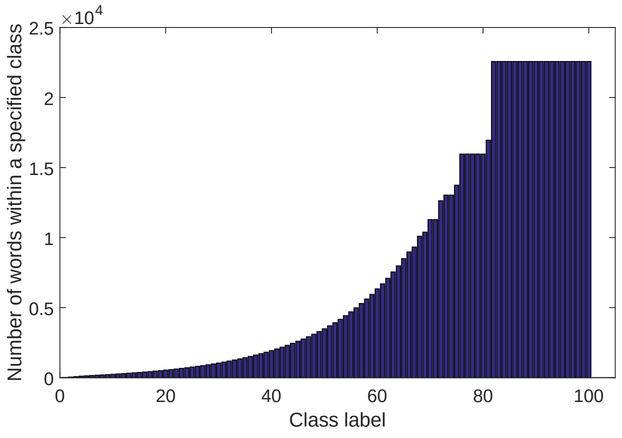


Figure 7.7: Class distribution for 128 million training words and 100 classes. Frequent words are in class label 0 and less frequent ones in class label 99

We expect that the class size should not affect the validation perplexity, as Mikolov stated [14]. In Figure 7.8, we observe a small improvement with the class size = 100 while the other size has a similar perplexity. However, we can conclude, according to Table 7.6, that an epoch lasts fewer minutes with class = 100 than = 600 or 1000, even though the number of epochs for 1 and 2 million words is larger with 100 than with 600 and 1000.

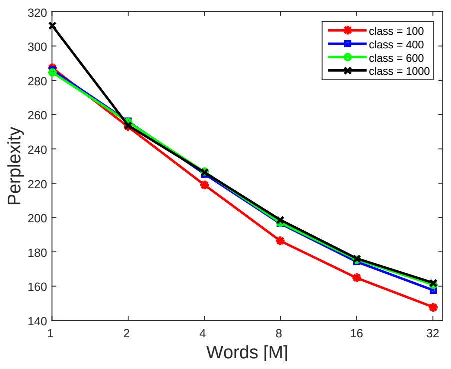


Figure 7.8: Validation perplexity with different class sizes.

To obtain the best accuracy the class size should be 1 [34], meaning each word in the vocabulary has its own class. For 1 million training words, the elapsed time is four days with 17 epochs

Words, in millions	Epochs			Minutes per epoch		
	Class:100	Class:600	Class:1000	Class:100	Class:600	Class:1000
1	23	17	15	7.2	11.5	16.6
2	23	21	20	15.7	24.0	35.0
4	22	20	20	36.3	47.4	52.1
8	21	20	17	45.8	44.8	79.6
16	20	19	18	138.8	197.0	276.6
32	19	17	17	292.7	284.4	336.7

Table 7.6: Number of epochs and minutes per epoch for three different class sizes: 100, 600, and 1000.

and the validation perplexity is 260. The relative reduction of perplexity with class = 100 is 9.4%, but the training time is far too long (4 days); thus we will not use a class of 1 in further measurements.

Time Steps

Time steps (or `bptt`) is a crucial parameter for RNN. Indeed, it represents the length of the context for a given word, and in an ideal case its value is infinite. Obviously, for practical reasons, this is infeasible, and so we carried out several experiments to observe its impact on validation perplexity. Also, we have already mentioned that a `bpttBlock` is used for practical implementation and causes the `bptt`'s effect to vanish. In order to have comparable results, in this particular section, `bptt` and `bpttBlock` are equal. Thus, the updates are done after processing `bptt` inputs, and the context length has the same value.

We can already imagine the effects of time steps according to this last statement. A large `bptt` catches a broad context, thus the model is improved because the network learns more word dependencies. On the contrary, with a small `bptt` the number of updates is large and so, since we learn more at every update, the model should again be improved. We expect a larger perplexity with a small `bptt`, because the context length is too narrow. But increasing `bptt` values will improve the model until a certain value. Then the perplexity would decrease, because the number of updates are too few.

Looking at Figure 7.9, we can clearly state that no significant improvements have been observed with a large `bptt` compared to a smaller one. The biggest difference between the two bars occurs in 2M; the absolute error between the black and green bars is -3.2 and represents 1.2% of relative improvement. On average, the relative improvement between the best and worst bar is approximately 0.5%, which is insignificant. Moreover, the training time is similar regardless of the time step's value. Also, note that the green and black bars are not displayed because a numerical error occurred for 128M. This issue occurs when the output probability is close to 0, and the logarithm function in Equation (3.8) then equals infinity. Usually, output probabilities should not equal 0. Unfortunately, with numerical precision and a large number of outputs, it might be the case.

Training time is also the same between all `bptt`—for small `bptt` values, the number of updates per epoch is larger than for large `bptts`; for example, with `bptt` equal to 5, the training algorithm does twice and three times more updates than with `bptt` equal to 10 and 15, respectively. So we need fewer epochs with small time step values, but with large values the training time per epoch is reduced since updates are less frequent. So, with small `bptt`, there are fewer epochs, but training time per epoch is long; with a large value, more epochs are needed but training time per epoch is reduced. In total, the overall training time is the same for all `bptts`.

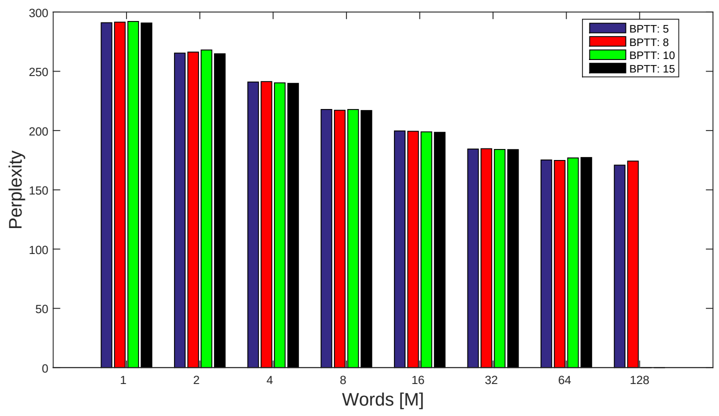


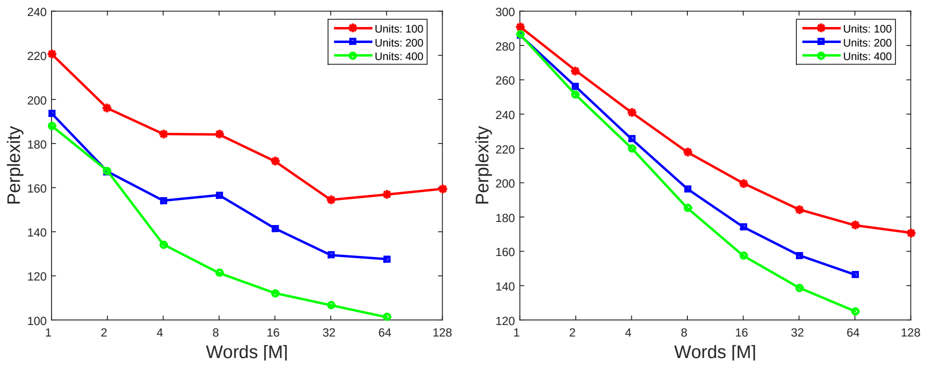
Figure 7.9: Validation perplexity with respect to `bptt` for 1M to 128M of words.

Since no significant improvements have been made with large `bptts`, we set the value to 5 except for some experiments in which we specify the `bptt`'s value.

Hidden Units

Hidden units are defined as the number of neurons in the hidden layer. It is a significant parameter for feedforward neural networks and for recurrent neural networks. Mikolov claims that the number of hidden units should represent the size of the vocabulary: few units for a small vocabulary and numerous units for a large vocabulary [13]. With few hidden units, underfitting can occur, and final perplexity deteriorates. On the other hand, too many hidden units leads to overfitting. However, finding the ideal hidden layer dimensionality is difficult. Because a generalized rule of thumb that finds the ideal layer size is impossible for all the problems, we tried different sizes for the hidden layer for the 1M to the 128M training set.

As expected, a high hidden layer dimension implies lower perplexity as it is shown in Figure 7.10 for training and validation. Indeed, hidden neurons are the output layer's input, more neurons implying a better hidden representation of a word and thus tending to reduce the perplexity. Additionally, we notice that the relative improvement between the hidden units = 100 and the hidden units = 200 is bigger than that between hidden units = 200 and hidden units = 400. Indeed, in Table 7.7 we observe that the relative perplexity tends to decrease while the dimension of hidden layer increases. At some point the number of hidden units will increase the perplexity, as we previously said. Also, while training data is increasing, the improvements are increasing too. For 2 million training data, we perform 1.9% better with 400 units than with 200 units, and for 64 million, this percentage goes up to 14.6%. The relative perplexity reduction could be even larger with 128 million or 792 million words.



(a) Training perplexity (b) Validation perplexity

Figure 7.10: Training and validation perplexity with three different numbers of hidden units.

Words, in millions	Relative PPL reduction [%]	
	Units: 100 & 200	Units: 200 & 400
1	1.7	-0.2
2	3.5	1.9
4	6.4	2.4
8	9.8	5.7
16	12.8	9.6
32	14.5	12.0
64	16.4	14.6

Table 7.7: Relative perplexity reduction between 100 and 200 hidden units and between 200 and 400.

Unfortunately, computing the perplexity for 256 million or larger proved impossible. Because the output computational complexity is H x C + H x W and the recurrent part is H x C + H x W, where H is the number of hidden units, C is the number of classes (Section 7.3.3) and W the number of words within a class. Once H gets larger then complexity increases and thus training time too. In particular, for 64 million and 400 units each epoch lasts-on averaged-1652 minutes (around 28 hours) and 18 epochs are needed to reach minima (Table 7.8). In total the training time was roughly 20 days; it becomes rapidly infeasible for a large vocabulary.

Since we were not able to train a large model with large hidden layer size, we tried to find our rule of thumb with respect to the baseline (Table 7.1). Figure 7.11 depicts the difference

Words, in millions	Epochs			Minutes per epoch		
	Unit: 100	Unit: 200	Unit: 400	Unit: 100	Unit: 200	Unit: 400
1	16	17	21	2.6	7.4	18.2
2	19	20	16	4.3	11.9	23.8
4	19	20	22	8.1	17.6	53.7
8	16	16	20	13.2	51.3	107.9
16	16	16	18	28.5	104.0	221.2
32	20	17	17	74.1	193.9	1228.0
64	17	16	18	186.5	495.2	1652.5
128	14	*	*	293.5	*	*

Table 7.8: The number of epochs to train RNN and the minutes needed to finish one epoch for 100, 200 and 400 hidden units. This symbol * means that a numerical error occurred.

between the validation perplexity of the baseline and diverse hidden layer size. Our idea is the following: when we double the size of the hidden layer, we reach a maximal value for a training data: for 100 units, the maximal value corresponds to 4 million, for 200 it is 8 million, and for 400 it is 16 million. Thus, for 800 units the maximal should be 32 million. In order to find a rule, we perform a polynomial interpolation of order 2 on the maximal difference between each hidden layer size: the maximum of blue, red, and green bars occurs at 4M, 8M, and 16M, and the values are 13.6, 31.5, and 43.9, respectively. We then interpolate those 3 points in order to obtain a second order equation. We chose a second order interpolation instead of a logarithmic function because we assume that, at a certain threshold, too many hidden units will penalize the validation perplexity because of overfitting. Thus perplexity must decrease like a second order function.

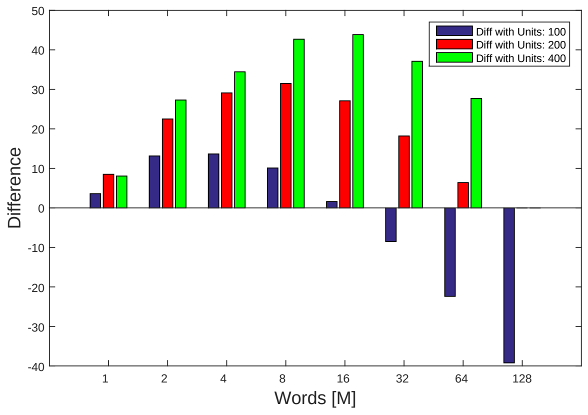


Figure 7.11: This graph represents the difference between baseline (Table 7.1) and the validation perplexity of Figure 7.10(b) with respect to number of words.

However, three points are too few to obtain a reliable equation, thus we repeat the experiment plotted in Figure 7.10 but with 50, 150, 250, and 300 units and only for 4, 8, and 16 million words. In the end, we obtain the following equation:

$$f(x) = -3.31215x^2 + 34.808x - 41.819, \tag{7.1}$$

where x should equal $log_2(\text{size training data})$, 2, 3, and 4 for 4, 8, and 16 million words, respectively. So, according to our rule, 32M with a hidden layer of dimension 800, we should obtain $f(log_2(32))$ equal to 49.42. Since this is the difference between the baseline and the RNN perplexity, the result is 126.5. With Equation (7.1), maximal value should be reached between 900 units and 1000 units. Unfortunately, we could not verify this rule because training 800 units with 32 million tokens takes too much time.

7.4 LSTM

All the following results are launched with the RWTHLM toolkit, because CURRENNT implementation could not handle a large input data size because of the NetCDF format (Section 6.2). Many options are available in order to build a network: the first layer should be a feedforward layer with an identity function or with a history of words (from 2 up to 9 words). There follows a recurrent layer: lstm or feedforward with different activation functions. The number of layers in the network is theoretically infinite, meaning an lstm and recurrent layer can be stacked.

7.4.1 Initialization

A Gaussian random distribution is applied in order to initialize weights, and so we tried to understand to what degree randomness influences LSTM networks. We launched 30 experiments with the same parameters and stopped them after 20 epochs. The sample standard deviation σ equals 2.380544 and the mean μ after 20 epochs for the 30 experiments is 192.56. So, if a new observation x is normally distributed with mean μ and standard deviation σ, then it ensures at 95% that x lies between $[\mu - 1.96\sigma, \mu + 1.96\sigma]$. In other words, every new measurement is contained in [187,9 , 197.2] with 95% of confidence for this particular LSTM's topology (Appendix B).

7.4.2 First Layer Type

The first layer, also called the projection layer, maps inputs to a continuous space [23]. First, we train the network including a projection layer and an LSTM layer with 300 units. We compare two possible projection layers: linear with identity activation function (first row of Table 7.9) and an n-gram feedforward layer (described in [6]), where n is equal to 2, 4, 5, 6, or 9. The projection layer size is 100 except for 6-gram and 9-gram where the dimension has to be incremented up to 300 and 450, respectively.

The performance of a n-gram feedforward layer is clearly worse than a simple linear layer according to Table 7.9. Also, we would expect a lower perplexity with a larger n compared

Layer type	Validation PPL	Time [hh:mm:ss]
Linear	171.4	7:03:58
FF 2-gram	173.3	9:00:39
FF 4-gram	183.5	8:29:05
FF 5-gram	188.4	8:24:50
FF 6-gram	192.4	17:52:31
FF 9-gram	203.4	19:02:15

Table 7.9: Perplexity and training time of projection layer type for 1 millions of training tokens after 15 epochs.

to a small n-gram. Because more words are taken into account. However, once n is growing the final perplexity is increasing too. Even training 9 more epochs, the final perplexity of the 9-gram is bigger than the 2-gram. The linear layer performs better than n-gram and it has a lower training time compared to all n-grams too. So for all further measurements we chose the linear layer with an identity activation function.

7.4.3 Stacked LSTM

Stacked LSTM outperforms standard networks for acoustic modeling [40]: multiple LSTM layers combine multiple representations of all preceding layers with a flexible context length [41]. So we wanted to see if similar improvements can be observed for language modeling.

2^{nd} layer units	Validation PPL	Time [hh:mm:ss]
0	171.4	7:03:58
40	171.3	7:24:17
100	171.6	7:45:31
200	172.9	9:24:24
400	173.2	17:22:25

Table 7.10: Perplexity and training time of 2-layer LSTM with 300 units for the first one and the second varies from 0 (one LSTM layer) 400.

In contrast to the findings of [40], the 2-layer LSTM does not improve significantly on a 1-layer LSTM. Indeed, as per Table 7.10, the difference between one and stacked LSTM layers is minuscule from 0.3 to 1.9 in absolute. Also, since stacking adds parameters, the training time gets longer with a large second LSTM layer. Thus, we will not use stacked LSTM for further experiments. We obtain similar performance with one LSTM layer in a shorter training time. Thus we use 1 LSTM layer for all further experiments.

7.4.4 Early Stopping

RWHTLM does not provide any early stopping method, thus stopping the training was done either by setting a maximum number of epochs or manually. To avoid such stopping methods, we modified the training in order to allow early stopping. If during 5 epochs the learning rate is small (less than 10¯4 or 10¯5) and there is not a significant improvement between the new validation perplexity and the old one (e.g., the difference between two perplexities is less than 1.0 or 0.5), the training ends. Additionally, the learning rate was halved if no improvement was observed.

The drawback of such early stopping is the saddle points; it happened that the gradient reached a saddle point. Thus the overall progress is slow, and if the learning rate is low, then our early stopping method ends the training. But we could reach better perplexity by increasing the learning rate, for example. However, our early stopping performs well in most cases as is depicted in Figure 7.12. Training ends after epoch 22 (blue vertical line) with a validation perplexity of 162.9265 and after 40 epochs the perplexity equals 162.7783. It is a relative improvement of 0.09%, which is insignificant.

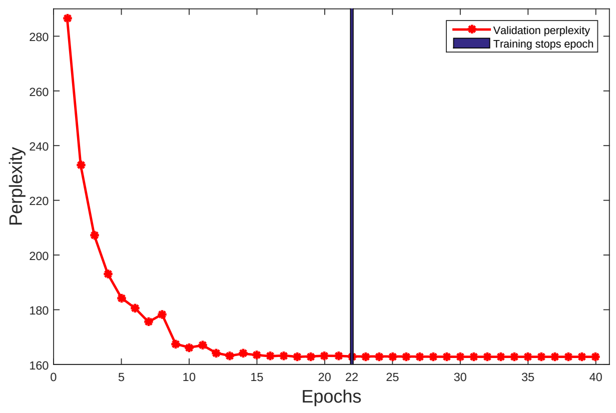


Figure 7.12: No improvement was observed if training continues for an infinite number of epochs.

7.4.5 Hyperparameters

In order to obtain the best recurrent model, we made several measurements in order to observe the effect of learning rate, batch size, and number of units.

Momentum

Momentum is added to the update equation in order to avoid *zigzagging*. This addition accelerates the gradient descent because it avoids an erratic behavior of the gradient. Also, the final perplexity should be similar than the one without momentum. According to Table 7.11 we notice that, as expected, the momentum term does not improve the validation perplexity . Additionally, we think that the number of steps will decrease with a μ greater than 0 since this avoids zigzagging. The second column of Table 7.11 shows that there is no significant differences between the number of epochs with $\mu = 0$ and the others, except with $\mu = 0.05$. For all further experiments no momentum term is used.

μ	Validation PPL	Epochs
0.00	164.6	27
0.05	165.0	24
0.10	165.8	26
0.25	165.7	25
0.50	166.2	24
0.75	164.9	27
0.90	165.6	28
1.00	166.4	27
1.10	165.3	26

Table 7.11: Perplexity and the number epochs needed to stop with respect to a momentum parameters.

Learning Rate

Table 7.12 summarizes the effect of a different starting learning rate η. With a small value such as 0.008, the performance is similar to other learning rates. The only difference lies in the number of epochs: more epochs are needed to reach the same perplexity (33 epochs instead of 28 on average), because each step in the cost function's minima direction is small with a low learning rate. On the other hand, a big η leads to bigger steps and so the minima should be reached in fewer epochs. However, large steps could skip the global minima and enter in a local minimum. This happens with a learning rate of 0.05, where the validation perplexity is 714.0 instead of 188.

η	Validation PPL	Epochs
0.008	189.2	33
0.010	189.1	29
0.015	188.6	29
0.020	187.6	27
0.025	187.5	29
0.050	714.0	23

Table 7.12: Perplexity and epoch according to several starting learning rates η for 1 million of training words.

Tuning learning rate is a problem, $\eta_k = \frac{1}{k}$, where k is the epoch, is commonly used for online learning ([42]) or a constant learning rate for online learning with mini-batch ([42]). However, we did not find a rule for LSTM and we also noticed that learning rate is highly correlated with other parameters such as the number of training tokens, the sequence (Section 7.4.5), and batch (Section 7.4.5).

Sequence

The sequence is used for training and perplexity evaluation; recall that we update LSTM's weights after processing a sequence S (Eq. (5.17)). Different methods could represent a sequence [36]:

1 `verbatim`: Sentence is considered as a sequence; no initialization is needed for the sequence size.

2 `concatenated`: Sequence is the number of consecutive sentences concatenated.

3 `fixed`: Group of consecutive words with a fixed length. The first words of the sentence could be in any position in the sentence, not necessary in the first one. RWTHLM sets this option by default.

For the two latter options, an initialization of sequence size is required, and the performance is similar. However *verbatim* performs significantly worse than the two other methods because it cannot learn word dependencies across sentences. Also, the internal memory of LSTM is reset every time the network has a new sequence as input [30]. Thus a small sequence size has small memory. However, the weights are updated after processing the whole sequence, so with a small length the network updates the weights more often. In comparison, a large sequence size is able to learn more word dependencies but there are less updates. We expect that varying the sequence size looks like a second order function. Indeed, increasing sequence length should decrease the perplexity until a certain threshold because more word dependencies are learnt. When the sequence size is larger than this threshold not enough updates are carried out and so the perplexity should increase. With Figure 7.13, we can observe this second order behavior and it seems that a minimum perplexity is around 200. The perplexity decreases from 25 to 200 and increases after 200.

However, we notice that the starting learning rate and sequence size are correlated as can be seen in Table 7.13. With a size of 600 or 1000 the results are poor with a learning rate larger than 0.002, but even with 0.002 smaller sequences perform better. The reason for this is that too few updates are done at every epoch. Also, for a sequence size of 100 and a learning rate of 0.008 we obtain the best result, whereas with a sequence size of 400 the optimal perplexity is obtained with a learning rate of 0.002.

Shuffling at the sequence level is recommended-in [36]-during the training and testing. Figure 7.14 confirms this since, even if the difference between the red and blue bars is small, it is not

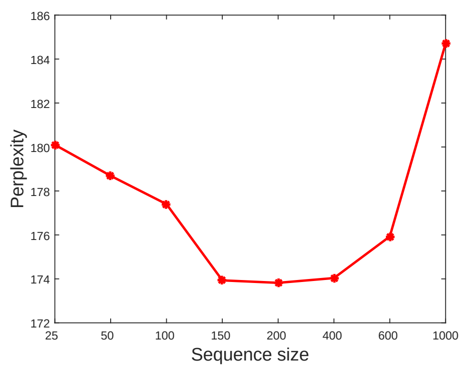


Figure 7.13: Validation perplexity for 1 million of word with a `fixed` varying sequence size and a starting learning rate = 0.002.

	Sequence size with `fixed` flag							
η	25	50	100	150	200	400	600	100
0.001	183.8	182.3	181.3	177.5	183.5	185.7	182.8	184.1
0.002	180.1	178.7	177.4	173.9	173.8	174.0	175.9	184.7
0.005	179.2	173.2	164.4	168.4	169.8	178.7	509.8	714.1
0.008	173.0	168.1	160.8	164.4	164.1	201.5	714.3	714.2
0.010	172.5	166.8	164.6	164.2	167.4	173.3	207.2	714.6
0.015	169.1	166.3	163.0	475.8	713.9	714.1	715.1	714.2

Table 7.13: Validation perplexity for 1 million of training words with respect to two variables: learning rate and sequence length. Several values are in red in order to emphasize that large sequence length leads to poor results.

a coincidence and due to the random initialization that the 8 first experiments and 9 out of 12 measurements are improved by shuffling. Also, we notice that there are differences within a same sequence length, but with a different starting learning rate. For sequence sizes of 100, 200, and 400 a learning rate smaller than 0.01 is required, whereas for 25 and 50 a learning rate bigger than 0.01 performs better. We previously mentioned that sequence size affects the overall perplexity in two ways: number of updates and the context. So we expected to observe a value that minimizes the perplexity. As per Figure 7.14, this value lies between 50 and 150 (with shuffling), and we think that random initialization introduced the small variations between 50, 100, and 150.

Batch

The batch size is the number of sequences that the network trains in parallel in order to speed up the training without penalizing the overall performance. Additionally, shuffling improved performance with respect to sequence size (Figure 7.14) so it might improve with respect to batch size too. As expected, training one epoch is 4 times faster with 16 batches than with 1 batch (Table 7.14) and needs less epochs to stop.

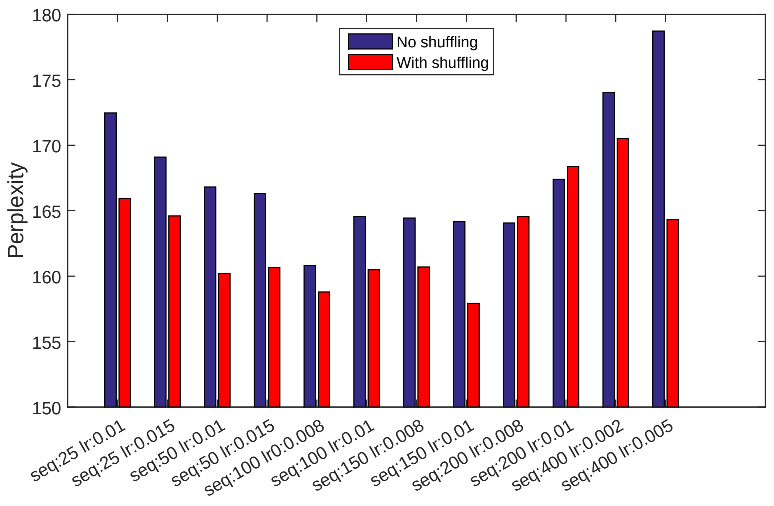


Figure 7.14: Comparison of the validation perplexity with and without shuffling sequences and with respect to sequence size and learning rate.

Batch	Epoch	Minutes per epoch
1	25	20.5
2	26	16.8
4	21	10.7
8	19	7.6
16	20	5.6

Table 7.14: Number of epoch and minutes per epoch with various batch size. No shuffling and 1 CPU are used.

Like the sequence, the batch size is correlated to the starting learning rate. Thus, as per Figure 7.15, we first set the learning rate to 0.01 and then decrease it according to the batch size (0.002 for 2, 4, and 8, and 0.001 for 16 batches) and also shuffle the sequence. Again shuffling improves the model, since curves with shuffling are below those without shuffling, especially with a learning rate of 0.001 and 4 and 8 batches. In comparison, 1 and 2 batches get the same perplexity, irrespective of shuffling. The best-achieved perplexity is with 4 batches, but we use 2 batches for all the following experiments because 4 batches behaves unstably with a larger learning rate; for example, with η = 0.01 the final perplexity is around 200 and 400 with η = 0.015.

First Layer Size

The first layer, or projection layer, is required to have a continuous representation of words as input for the LSTM layer [23]. With a small number of units, the data are not representative

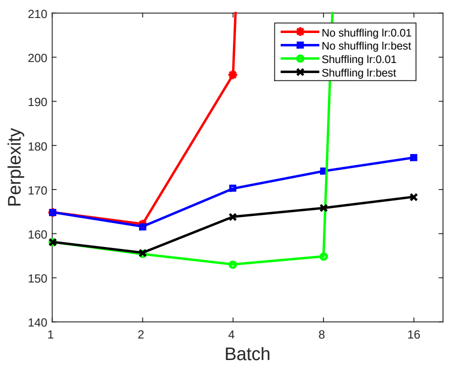


Figure 7.15: Batch size with default (i.e. 0.01), best (i.e. lowest perplexity) learning rate and with/without shuffling.

enough, thus leading to the poor result. However, increasing layer-size we expect better outcomes and a significantly higher training time. With Table 7.15, we indeed obtain better results with 200 or 300 units, even though the difference is not significant between 50, 200, and 300. Regarding the time, 50,100, and 150 have similar training times while 200 and 300 are longer, as expected.

Projection layer size	Valid PPL	Minutes per epoch
10	180.3	24.3
50	166.5	19.5
100	171.4	18.3
150	172.7	18.6
200	163.4	51.1
300	167.0	61.4

Table 7.15: Perplexity and minutes per epoch for 1 million training tokens and various first layer units. The number of epochs is fixed to 15.

LSTM Units

The number of units in the LSTM layer had the same issue as the number of neurons in the hidden layer for RNN. Too many LSTM units leads to overfitting the data, and too few leads to underfitting. In both cases, the validation perplexity is penalized. Thus, we should find the ideal number of units. Obviously this depends on the number of training tokens: small training sets do not need a large LSTM size as large sets do like for RNN. However a large training set and a large number of LSTM units was very time-consuming, thus we only state results for 1 million tokens in Table 7.16. We obtain the lowest perplexity with 100 units, and—as we expected—with fewer units the result is not as good as with a large number of units.

LSTM-units	Validation PPL	Training time [hh:mm:ss]
40	176.7	1:19:04
100	166.2	3:24:18
200	166.1	9:05:56
300	169.5	7:44:26
400	172.2	13:40:19

Table 7.16: Perplexity and training time with respect to the dimension of LSTM for a small training set (1 million words).

7.5 Testing

After selecting the optimal parameters for RNN and LSTM, we observe the overall quality of our trained networks. Thanks to Ivan Slijepčević [1], who provided results for feedforward neural networks (FFNN [6]), we can compare RNN and LSTM, and both with FFNN and the baseline. Ivan, however, applied a minimal count of 3 on his training set; thus we subdivided the testing section into two parts: with and without minimal count. Note that, we previously use a validation set for tuning all parameters of RNN and LSTM. In this section, a test set is used.

7.5.1 Minimal count 3

With Figure 7.16, looking at the perplexity for 1 and 2 million we notice that they have roughly the same values: 200 and 204 for baseline, 178 and 176 for RNN, and 156 and 157 for LSTM. This holds except for FFNN where 2 million performs significantly worse than 1 million. We would expect a lower perplexity a for larger training set, but as we previously mentioned in Section 3.2.1, the `<unk>` token is too frequent and becomes more predictable, so perplexity decreases. RNN significantly improves language modeling compared to baseline for 1 and 2 million (10.7% and 13.8% better, respectively) and achieved perplexity is significantly lower for 2 million compared to FFNN (10.4%). However, RNN is slightly worse than FFNN for 1M (-1.9%). Perhaps, the RNN's parameters are not optimal for this training set. Additionally, LSTM outperforms our optimal RNN for both 1 and 2 million training tokens and thus the KN5. The relative improvement of LSTM over RNN and KN5 is about 13% and 22%, respectively, for 1 million and 10% and 23%, respectively, for 2 million (Table 7.17).

Words, in millions	Relative PPL reduction [%]		
	KN5	FFNN	RNN
1	22.0	11.0	12.7
2	23.1	20.1	10.8

Table 7.17: Improvement on test set in percentage of LSTM over baseline, feedforward neural network and RNN with a minimal count of 3.

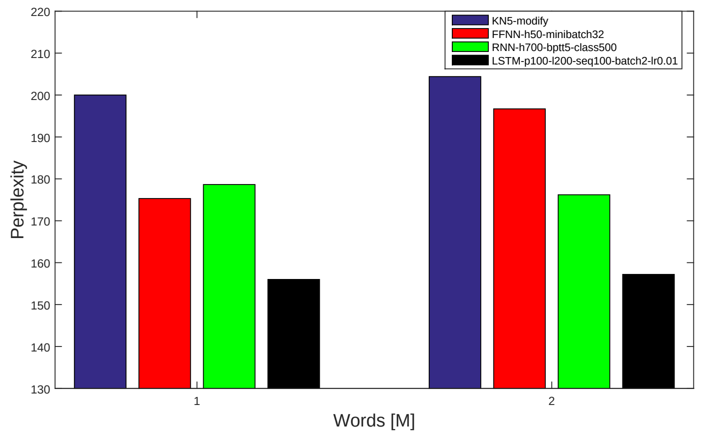


Figure 7.16: For 1 and 2 millions of training word, comparison of test perplexity for KN5, feedforward neural network, RNN and LSTM with optimal parameters. All words occurring less than 3 times are mapped to `<unk>` token.

Although LSTM outperforms older techniques, we compare how long the training is for all models. Training time for all networks is far from that of KN5, which is less than 25 seconds. To our knowledge, it is impossible to beat 30 seconds with such complex architectures as FFNN, RNN, and LSTM. In Table 7.18, the number of epochs needed to finish training is 20 for LSTM, whereas only 13 and 10 are required for FFNN (for 1 and 2 million words, respectively), and 18 and 16 for RNN. However, the minutes per epoch is significantly lower for LSTM, 9 and 53 minutes (for 1 and 2 million, respectively), 52 and 168 for FFNN, 72 and 120 for RNN. Even if more epochs are needed, the total LSTM training time is lower than FFNN and RNN,. We should mention that Ivan obtained FFNN results using MATLAB, whereas for RNN and LSTM C and C++ were used. Moreover, since MATLAB is considerably slower than C and C++, this penalizes the overall training time for FFNN. Last but not least, LSTM used 2 CPUs because we set the batch size to 2 and only RNN ran under 1 CPU. Thus, it could explain why there is such huge difference with RNN's training time and LSTM's one.

	KN5-modified		FFNN		RNN		LSTM	
Words, in millions	Epoch	min. per epoch	Epoch	min. per epoch	Epoch	min. per epoch	Epoch	min. per epoch
1	1	0.3	13	52.2	18	72.3	20	9.3
2	1	0.3	10	168.2	16	120.5	20	53.1

Table 7.18: Number of epoch required to stop training and minutes per epoch for all models and with a minimal count of 3.

7.5.2 No minimal count

In this section, we set minimal count to 0, meaning that all training words were taken into account. We expect a lower perplexity compared to measurements with a minimal count of 3. This because, without any minimal count, the training model took into account all words occurring once or twice, and many of them are miswritten, or are tokenization errors, foreign words, or punctuation. Moreover, in order to compute validation and test perplexity, we discard all words that occur in the validation or test set but that are not part of training vocabulary.

As shown in Figure 7.17, we tried two different RNN topologies, one with 400 hidden units and the other with 800, to see if there was a significant difference. Obviously the results are similar, only a small improvement of 1.5%, 1.9%, and 1.5% for 1, 2, and 4 million words with a hidden layer of dimension 800 compared to a hidden layer of 400 units. Both topologies outperform the standard KN5 as it is shown in Table 7.19, especially for 8 million: the relative perplexity reduction is about 19% for RNN-h400. Again, the LSTM outperforms both RNNs and, so, the baseline, and in Table 7.20 we see that the relative improvements of LSTM over KN5 are outstanding. For 8 million it is 31.2% better, and knowing the fact that KN5 has performed well during the last few decades, we can conclude that LSTM is the new architecture to model languages regarding overall performance.

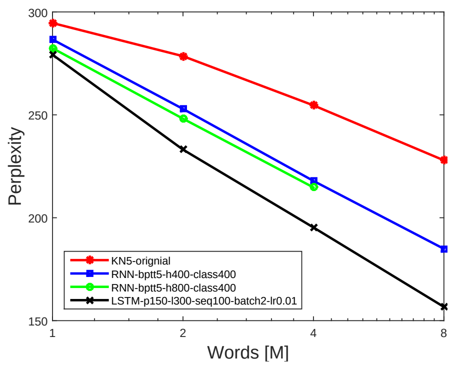


Figure 7.17: Comparison of achieved perplexity on test set for KN5, two RNNs and LSTM with optimal parameters and taking into account all words in training set.

Words, in millions	Relative PPL reduction [%] RNN-h400	RNN-h800
1	2.7	4.2
2	9.2	10.9
4	14.4	15.7
8	18.9	*

Table 7.19: Improvement on test set in percentage of the two RNNS over baseline without a minimal count.

Words, in millions	Relative PPL reduction [%]		
	KN5	RNN-h400	RNN-h800
1	5.2	2.6	1.1
2	16.3	7.8	6.1
4	23.3	10.4	9.1
8	31.2	15.3	*

Table 7.20: Improvement on test set in percentage of LSTM over baseline and the two RNNs without a minimal count.

In Table 7.21, we notice that KN5 is much faster than any other architecture. It took 1 minute and 10 seconds to train with 8 million words, whereas this figure is 36 hours for RNN-h400 and 152 hours for LSTM. As expected, training RNN-h800 took longer than RNN-h400 since the computation is highly dependent on the number of hidden units (Section 7.3.3). With a minimal count of 3, training LSTM was faster than RNN; however, with a minimal count of 0, LSTM took longer to finish training compared to RNN-h400. The reason for this is the following: RNN's complexity for language modeling depends on hidden units and class, whereas LSTM has a complexity of $O(W)$, where W are the total weights of the network [30], and depend on input and output layer (vocabulary size), projection layer, and LSTM unit numbers. Thus increasing vocabulary size penalizes training time for LSTM but not for RNN. In order to speed up LSTM, a minimal count is needed if we assume that words occurring less than 3 times are mistakes, tokenization, or punctuation errors. It is a valid assumption: here are some examples of words occurring once:

- *Miswritten*: cluelessly, concevable
- *Tokenization*: 's-Roosevelt, –Pittsburgh, 258kg, www.csrees.usda.gov
- *Punctuation*: ...., !*#
- *Foreign languages*: classé, cul-de-sac

	KN5-modified		RNN-h400		RNN-h800		LSTM	
Words, in millions	Epoch	min. per epoch	Epoch	min. per epoch	Epoch	min. per epoch	Epoch	min. per epoch
1	1	0.1	21	18.2	20	34.6	21	43.3
2	1	0.2	16	27.5	21	159.2	19	92.3
4	1	0.7	22	53.7	20	311.9	22	197.7
8	1	1.1	20	107.9	*	*	22	414.4

Table 7.21: Training time for 3 different networks and the baseline with the full vocabulary.

8 Conclusion

We have shown that recurrent neural networks (RNNs) improve language modeling over KN5 baseline on one billion word benchmark database regardless of minimal count. The major drawback of such networks is that they are time-consuming and cannot learn long-term dependencies. Then we introduced a new architecture called LSTM, which learned long-term dependencies thanks to its internal memory. As expected LSTM outperforms all previous structures; in particular, LSTM reduced perplexity by 25% over baseline and 15% over RNN with 400 hidden units (8 million training words and no minimal count). Moreover, if we set a minimal count of 3 then the improvements are significant: 22% over baseline, 20% over FFNN, and 11% over RNN (2 million training words). Additionally, we think that our parameters are not optimal for large vocabulary such as 64M and more. But tuning the parameters is long for large vocabulary, especially when the training time is longer than two weeks.

Unfortunately, we could not compare LSTM, RNN, and KN5 for more than 8 million training words, because with a large dimension layer training time for neural networks could last several days or weeks. For example, as per Table 7.8 with 400 hidden units and 64M training words, the network converged in 3 weeks. LSTM performs very well, but for small vocabularies, so the next step will be to speed up the training. For example, using a GPU instead of a CPU, since they are more efficient. Nevertheless, we were not able to use a GPU for our measurements, but we know that CURRENNT (Section 6.2) uses LSTM with a GPU, but input files were too big and thus we could not run LSTM with CURRENNT. However, it might be interesting to adapt CURRENNT for language modeling purposes. Also, we saw that a new LSTM toolkit available for GPUs is going to be integrated into Kaldi[1].

Last but not least, LSTM and RNN learn words from the left to right, but we think that we can also use the words at the right position to obtain more meaningful information. One solution is bidirectional LSTM or RNN, and this has already been used for acoustic modeling [21, 43] and outperforms a Gaussian mixture model (GMM) and deep networks on the Wall Street Journal database.

[1] https://github.com/dophist/kaldi-lstm

A Training derivation

For the following sections, the derivations as well as the equations are described in multiple papers [44, 39, 45]. The reader can look at them in order to learn more about backpropagation and its implementation.

A.1 Usual Backpropagation

All information about the derivation and function for the forward pass will be explained in this section and for the backpropagation and the backpropagation throuhgh time in the following section.

t	:	time index
V	:	Vocabulary size
N	:	Number of words
H	:	Number of hidden units in the hidden layer
R	:	Number of recurrent units in the recurrent layer (should be equal to H)
β	:	regularization parameter
η	:	learning rate
$\mathbf{w}_t$	:	input vector at time t of size V x 1
$\mathbf{s}(t)$	:	vector containing hidden units value at time t of size H x 1
$\mathbf{s}(t-1)$	:	vector containing recurrent units value at time t of size H x 1
$\mathbf{y}_t$	:	output vector at time t containing the probability of the next word of size V x 1
$\mathbf{d}_t$	:	desired output, i.e. the expected word at time t+1. Size is V x 1
$\mathbf{e}_o(t)$	:	error vector at the output layer of size V x 1
$\mathbf{e}_h(t)$	:	error vector at the hidden layer of size H x 1
$\mathbf{U}$	:	weight matrix between input and hidden layer of size H x V
$\mathbf{W}$	:	weight matrix between recurrent and hidden layer of size H x H
$\mathbf{V}$	:	weight matrix between hidden and output layer of size V x H

$$\begin{aligned}
\text{Cost function: } C &= \sum_{n=1}^{N}\sum_{v=1}^{V} d_{nv} log\left(\frac{1}{y_{nv}}\right) + (1-d_{nv}) log\left(\frac{1}{1-y_{nv}}\right) && \text{(A.1)}\\
\text{sigmoïd: } f(x) &= \frac{1}{1-e^{-x}} && \text{(A.2)}\\
f'(x) &= f(x)\big(1-f(x)\big) && \text{(A.3)}\\
\text{softmax: } g(x_l) &= \frac{e^{x_l}}{\sum_n e^{x_n}} && \text{(A.4)}\\
g'(x_l) &= g(x_l)\big(1-g(x_l)\big) && \text{(A.5)}
\end{aligned}$$

Let us define a new variable called *net* that is the linear combination of the inputs

$$\begin{aligned}
\text{input to hidden layer: } net_j(t) &= \sum_{v=1}^{V} u_{jv} w_v(t) + \sum_{h=1}^{H} w_{jh} s_h(t-1) && \text{(A.6)}\\
\text{hidden to output layer: } net_k(t) &= \sum_{h=1}^{H} v_{kh} s_h(t) && \text{(A.7)}
\end{aligned}$$

To train and so update all weight matrices, we apply a gradient descent algorithm. To do so, we need the derivative of the C with respect to a specific weight matrix. For all derivations we will abbreviate the gradient as ∇ and $\nabla_{\mathbf{w}} C$ is the gradient of C at $\mathbf{w}$. Using the chain rule, $\nabla_{\mathbf{w}} C$ is easily computed.

$$\nabla_{\mathbf{w}} C = \nabla_{\mathbf{net}} C \, \nabla_{\mathbf{w}} net \tag{A.8}$$

The left term is easily compute since *net* is a linear combination of the weight matrix whereas the right term depends on which layer it is computed. At the output layer :

$$\nabla_{\mathbf{net_{nv}}} C = \nabla_{\mathbf{y_{nv}}} C \, \nabla_{\mathbf{net_{nv}}} y_{nv} = \big(y_{nv} - d_{nv}\big) \tag{A.9}$$

Notice that it is a special case; combining either softmax or sigmoïd with cross-entropy cost function allows us to write $\nabla_{\mathbf{net}} C$ for the output nodes as above. Also note that $\nabla_{\mathbf{net}} C$ can be considered as an error [45]. Regarding the hidden layer, the error is more tricky and can be written as :

$$\begin{aligned}
\nabla_{\mathbf{net_{nh}}} C &= \sum_{v=1}^{V} \big(\nabla_{y_{nv}} C \, \nabla_{net_{nv}} y_{nv} \, \nabla_{s_{nh}} net_{nv}\big) \, \nabla_{s_{nh}} net_{nh} && \text{(A.10)}\\
&= \sum_{v=1}^{V} \nabla_{net_{nv}} C \; w_{vh} f'(net_{nh}) && \text{(A.11)}
\end{aligned}$$

Then the three matrices **U**, **V** and **W** changes according to the term $-\eta\nabla_{\mathbf{U}}C$, $-\eta\nabla_{\mathbf{U}}C$ and $-\eta\nabla_{\mathbf{U}}C$.

$$-\eta\nabla_{\mathbf{V_{vh}}}C = -\eta\sum_{n=1}^{N}\nabla_{\mathbf{net_{nv}}}C\, s_{nh} \tag{A.12}$$

$$-\eta\nabla_{\mathbf{U_{hv}}}C = -\eta\sum_{n=1}^{N}\nabla_{\mathbf{net_{nh}}}C\, w_{pv} \tag{A.13}$$

$$-\eta\nabla_{\mathbf{W_{hr}}}C = -\eta\sum_{n=1}^{N}\nabla_{\mathbf{net_{nh}}}C\, s_{pr}(t-1) \tag{A.14}$$

A.2 Backpropagation through Time

Recall the unfolded recurrent neural network figure (4.2) where τ is the number of time steps. Now equation (A.11) should be propagated backward :

$$\nabla_{net_{nj}}C(t-1) = \sum_{h=1}^{H}\nabla_{net_{nh}}C(t)w_{hj}f'\big(s_{nj}(t-1)\big) \tag{A.15}$$

Notice that it is recursively computed. Now the equation (A.11) has been rewritten, the update of **U**, **V** and **W** is modified. Let us defined error in the output layer ($\mathbf{e_o}(t)$) and $\mathbf{e_h}(t)$ is the error propagated from the output to hidden layer.

$$\mathbf{e_o}(t) = \mathbf{y}(t)-\mathbf{d}(t) \tag{A.16}$$

$$\mathbf{e_h}(t) = d_h\big(\mathbf{e_o}(t)^T\mathbf{V}, t\big) \tag{A.17}$$

Where $\mathbf{d}(t)$ is the desired vector and $d_h(x,t)$ is the equation (4.6). The new matrix between the output and hidden layer is:

$$\mathbf{V}_{hv}(t+1) = \mathbf{V}_{hv}(t)-\eta s_h(t)e_{ov}(t) \tag{A.18}$$

Backpropagating the error through the unfolded network is done recursively (cf eq (A.15)) and can be rewritten in a more readable way:

$$\mathbf{e}_h(t-\tau-1) = d_h(\mathbf{e}_h(t-\tau)\mathbf{W}, t-\tau-1) \tag{A.19}$$

The two last matrices are updating as follow :

$$\mathbf{U}_{ij}(t+1) = \mathbf{U}_{ij}(t)+\eta\sum_{z=0}^{\tau}w_i(t-z)\,e_{hj}(t-z) \tag{A.20}$$

$$\mathbf{W}_{lj}(t+1) = \mathbf{W}_{lj}(t)+\eta\sum_{z=0}^{\tau}s_l(t-z)\,e_{hj}(t-z) \tag{A.21}$$

B Parameters

This appendix contains all parameters for all figures and tables of this master thesis.

B.1 RNN

	Figure 6.1: Kaldi	Figure 6.1: Mikolov	Figure 6.2	Figure 6.3: β_{scaled}	Figure 6.3: $\beta_{notScaled}$	Figure 6.3: Mikolov
MinCount	0	0	0	0	0	0
Class	0	150	0	0	0	400
β	10^{-7}	10^{-7}	10^{-7}	10^{-6}	10^{-7}	10^{-6}
η	0.1	0.1	0.1	0.1	0.1	0.1
BPTT	5	5	5	5	5	5
Hidden Units	100	100	100	100	100	100
ME order	0	0	0	0	0	0
ME hash size	0	0	0	0	0	0
Train set	varying	varying	2M	varying	varying	varying

Table B.1: Parameters of results in Section 6.2.1 "Kaldi Issues".

	Figure 7.1	Table 7.3: 10 Threads	Table 7.3: 1 Thread	Figure 7.2
MinCount	0	0	0	0
Class	0	0	400	0
β	10^{-7}	10^{-7}	10^{-7}	10^{-7}
η	0.1	0.1	0.1	0.1
BPTT	4	5	5	4
Hidden Units	50	200	200	100
ME order	0	5	5	5
ME hash size	0	100M	100M	500M
Train set	2M	varying	varying	4M

Table B.2: Parameters of Section 7.3.1 "Multi-Threading"

	RNN+ME	RNN
MinCount	0	0
Class	400	400
β	10^{-7}	10^{-7}
η	0.1	0.1
BPTT	5	5
Hidden Units	200	200
ME order	5	0
ME hash size	10M	0
Train set	varying	varying

Table B.3: Parameters of Figure 7.3 and Table 7.5. For Table 7.4, same parameters as RNN+ME except ME hash is 500M

	Figure 7.4: η	Figure 7.5: η decay	Figure 7.6: β	Figure 7.8, Table 7.6: Class	Figure 7.9: BPTT	Figures 7.10, 7.11, Tables 7.7, 7.8: Hidden Units
MinCount	0	0	0	0	0	0
Class	400	400	400	varying	400	400
β	10^{-7}	10^{-7}	varying	10^{-7}	10^{-7}	10^{-7}
η	varying	0.2	0.1	0.1	0.1	0.1
BPTT	6	6	5	5	varying	5
Hidden Units	100	200	100	200	100	varying
ME order	5	5	0	0	0	0
ME hash size	10M	10M	0	0	0	0
Train set	2M	varying	varying	varying	varying	varying

Table B.4: Parameters for all hyperparameters for RNN measurements.

B.2 LSTM

	Section 7.4.1	Table 7.9
MinCount	3	3
Max. Epoch	20	15
Sequence	100	100
η	0.01	0.01
1st layer	10	100
2nd layer	40	300
3rd layer	0	0
1st layer type	linear-id.	varying
Momentum	0	0
Shuffling	None	None
Batch	1	1
Train set	1M	

Table B.5: Parameters of Randon initialization (second column) and The first layer type (third column)

	Table 7.10	Figure 7.12	Table 7.11	Table 7.12
MinCount	3	3	3	3
Max. Epoch	15	200	50	50
Sequence	100	100	100	100
η	0.01	0.01	0.01	varying
1st layer	100	50	50	10
2nd layer	300	100	100	40
3rd layer	varying	0	0	0
1st layer type	linear-id.			
Momentum	0	0	varying	0
Shuffling	None			
Batch	1	1	1	1
Train set	1M			

Table B.6: Parameters of 2nd LSTM layer size (second column), early stopping (third column), momentum (fourth column) and learning rate (last column)

	Figure 7.13	Table 7.13	Figure 7.14	Table 7.14	Figure 7.15
MinCount	3	3	3	3	3
Max. Epoch	50	50	50	50	50
Sequence	varying	varying	varying	100	100
η	0.002	varying	varying	0.01	varying
1^{st} layer	50	50	50	100	100
2^{nd} layer	100	100	100	200	200
3^{rd} layer	0	0	0	0	0
1^{st} layer type			linear-id.		
Momentum	0	0	0	0	0
Shuffling	None	None	Yes/No	No	Yes/No
Batch	1	1	1	varying	varying
Train set			1M		

Table B.7: Parameters of Section Sequence 7.4.5 (second to fourth column) and Section Batch 7.4.5 (fifth and sixth column)

	Table 7.15	Table 7.16
MinCount	3	3
Max. Epoch	15	50
Sequence	100	100
η	0.01	0.02
1^{st} layer	varying	100
2^{nd} layer	300	varying
3^{rd} layer	0	0
1^{st} layer type	linear-id.	linear-id.
Momentum	0	0
Shuffling	None	None
Batch	1	1
Train set	1M	

Table B.8: Parameters of First layer size (second column) and First LSTM size (third column)

B.3 Testing

	RNN-h700	LSTM-p100	RNN-h400	RNN-h800	LSTM-p150
MinCount	3	3	0	0	0
Max. Epoch	100	50	100	100	50
Sequence	None	100	None	None	100
Class	400	None	400	400	None
BPTT	5	None	5	5	None
η	0.1	0.01	0.1	0.1	0.01
1st layer	700	100	400	800	150
2nd layer	0	200	0	0	300
3rd layer	0	0	0	0	0
1st layer type	Recurrent	linear-id.	Recurrent	Recurrent	linear-id.
Momentum	0	0	0	0	0
Shuffling	No	Yes	No	No	Yes
Batch	1	2	1	1	2
Train set			varying		

Table B.9: The second and third columns are parameters for Testing section with a minimal count of 3 (Figure 7.16, Tables 7.17,7.18). Instead the three last columns contain the parameter without minimal count (Figure 7.17 and Tables 7.20,7.19 and 7.21). The recurrent layer type refers to hidden layer for RNN and its size is in row with variable: 1st layer.

Bibliography

[1] I. Slijepčević, "Improving learning efficiency in large scale neural networks," Master's thesis, École Polytechnique Fédérale de Lausanne, 2014.

[2] T. Mikolov, *Statistical language Model Based on Neural Networks*. PhD thesis, Brno University of Technology, 2012.

[3] S. Katz, "Estimation of probabilities from sparse data for the language model component of a speech recognizer," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 35, no. 3, pp. 400–401, 1987.

[4] R. Kneser and H. Ney, "Improved backing-off for m-gram language modeling," in *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, vol. 1, pp. 181–184, IEEE, 1995.

[5] S. F. Chen and J. Goodman, "An empirical study of smoothing techniques for language modeling," in *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pp. 310–318, Association for Computational Linguistics, 1996.

[6] Y. Bengio, R. Ducharme, and P. Vincent, "A neural probabilistic language model," *Journal of Machine Learning Research*, vol. 3, pp. 1137–1155, 2003.

[7] J. L. Elman, "Finding structure in time," *COGNITIVE SCIENCE*, vol. 14, no. 2, pp. 179–211, 1990.

[8] H. Schwenk and J.-L. Gauvain, "Connectionist language modeling for large vocabulary continuous speech recognition," in *Acoustics, Speech, and Signal Processing (ICASSP), 2002 IEEE International Conference on*, vol. 1, pp. I–765, IEEE, 2002.

[9] H. Schwenk and J.-L. Gauvain, "Neural network language models for conversational speech recognition.," in *INTERSPEECH*, 2004.

[10] A. Emami and F. Jelinek, "Exact training of a neural syntactic language model," in *Acoustics, Speech, and Signal Processing, 2004. Proceedings.(ICASSP'04). IEEE International Conference on*, vol. 1, pp. I–245, IEEE, 2004.

[11] H. Schwenk and J.-L. Gauvain, "Building continuous space language models for transcribing european languages.,"

[12] H. Schwenk and J.-L. Gauvain, "Training neural network language models on very large corpora," in *Proceedings of the Conference on Human Language Technology and Empirical Methods in Natural Language Processing*, HLT '05, (Stroudsburg, PA, USA), pp. 201–208, Association for Computational Linguistics, 2005.

[13] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur, "Recurrent neural network based language model.," in *INTERSPEECH*, pp. 1045–1048, 2010.

[14] T. Mikolov, S. Kombrink, L. Burget, J. Cernocky, and S. Khudanpur, "Extensions of recurrent neural network language model," in *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pp. 5528–5531, IEEE, 2011.

[15] T. Mikolov, A. Deoras, D. Povey, L. Burget, and J. Cernocky, "Strategies for training large scale neural network language models," in *Automatic Speech Recognition and Understanding (ASRU), 2011 IEEE Workshop on*, pp. 196–201, IEEE, 2011.

[16] Y. Bengio, J.-S. Senécal, *et al.*, "Quick training of probabilistic neural nets by importance sampling," in *AISTATS Conference*, 2003.

[17] H. Schwenk, "Continuous space language models," *Computer Speech & Language*, vol. 21, no. 3, pp. 492–518, 2007.

[18] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *Neural Networks, IEEE Transactions on*, vol. 5, pp. 157–166, Mar 1994.

[19] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, pp. 1735–1780, Nov. 1997.

[20] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with lstm," *Neural computation*, vol. 12, no. 10, pp. 2451–2471, 2000.

[21] H. Sak, A. Senior, and F. Beaufays, "Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition," *CoRR*, vol. abs/1402.1128, 2014.

[22] A. Graves, A.-R. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 6645–6649, May 2013.

[23] M. Sundermeyer, R. Schlüter, and H. Ney, "Lstm neural networks for language modeling.," in *INTERSPEECH*, 2012.

[24] D. Soutner and L. Müller, "Application of lstm neural networks in language modelling," in *Text, Speech, and Dialogue*, pp. 105–112, Springer, 2013.

[25] J. T. Goodman, "A bit of progress in language modeling extended version," *Microsoft Research: technical report*, no. MSR-TR-2001-72, 2001.

[26] M. Hermans and B. Schrauwen, “Training and analysing deep recurrent neural networks,” in *Advances in Neural Information Processing Systems 26* (C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, eds.), pp. 190–198, 2013.

[27] M. Minsky and S. Papert, “Perceptrons: An introdcution to compuational geometry.” MIT Press, 1969.

[28] T. Alumäe and M. Kurimo, “Efficient estimation of maximum entropy language models with n-gram features: an srilm extension.,” in *INTERSPEECH*, pp. 1820–1823, 2010.

[29] F. A. Gers, N. N. Schraudolph, and J. Schmidhuber, “Learning precise timing with lstm recurrent networks,” *The Journal of Machine Learning Research*, vol. 3, pp. 115–143, 2003.

[30] A. Graves and J. Schmidhuber, “Framewise phoneme classification with bidirectional lstm networks,” in *Neural Networks, 2005. IJCNN’05. Proceedings. 2005 IEEE International Joint Conference on*, vol. 4, pp. 2047–2052, IEEE, 2005.

[31] F. A. Gers, *Long Short-Term Memory in Recurrent Neural Networks.* PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 2001.

[32] C. Chelba, T. Mikolov, M. Schuster, Q. Ge, T. Brants, P. Koehn, and T. Robinson, “One billion word benchmark for measuring progress in statistical language modeling,” in *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[33] A. Stolcke *et al.*, “Srilm-an extensible language modeling toolkit.,” in *INTERSPEECH*, 2002.

[34] T. Mikolov, S. Kombrink, A. Deoras, L. Burget, and J. Cernocky, “Rnnlm-recurrent neural network language modeling toolkit,” in *Proc. of the 2011 ASRU Workshop*, pp. 196–201, 2011.

[35] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz, J. Silovsky, G. Stemmer, and K. Vesely, “The kaldi speech recognition toolkit,” in *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*, IEEE Signal Processing Society, Dec. 2011. IEEE Catalog No.: CFP11SRW-USB.

[36] M. Sundermeyer, R. Schlüter, and H. Ney, “rwthlm–the rwth aachen university neural network language modeling toolkit,” in *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[37] F. Weninger, J. Bergmann, and B. Schuller, “Introducing currennt–the munich open-source cuda recurrent neural network toolkit,” *Journal of Machine Learning Research*, vol. 15, 2014.

[38] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” *arXiv preprint arXiv:1301.3781*, 2013.

[39] M. Bodén, "A guide to recurrent neural networks and backpropagation," *The Dallas project, SICS technical report*, 2002.

[40] A. Graves, A.-R. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 6645–6649, IEEE, 2013.

[41] X. Li and X. Wu, "Constructing long short-term memory based deep recurrent neural networks for large vocabulary speech recognition," *arXiv preprint arXiv:1410.4281*, 2014.

[42] C. M. Bishop *et al.*, "Neural networks for pattern recognition," 1995.

[43] A. Graves, N. Jaitly, and A.-R. Mohamed, "Hybrid speech recognition with deep bidirectional lstm," in *Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on*, pp. 273–278, IEEE, 2013.

[44] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[45] J. Guo, "Backpropagation through time," 2013.