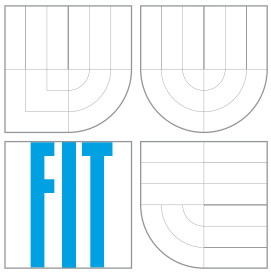
BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

RECURRENT NEURAL NETWORKS WITH ELASTIC TIME CONTEXT IN LANGUAGE MODELING

REKURENTNÍ NEURONOVÉ SÍTĚ S PRUŽNÝM ČASOVÝM KONTEXTEM V JAZYKOVÉM MODELOVÁNÍ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE

Bc. KAREL BENEŠ

SUPERVISOR
VEDOUCÍ PRÁCE

Dipl.-Ing MIRKO HANNEMANN,

BRNO 2016

Master's Thesis Specification/18955/2015/xbenes20

Brno University of Technology - Faculty of Information Technology

Department of Computer Graphics and Multimedia — Academic year 2015/2016

Master's Thesis Specification

For: **Beneš Karel, Bc.**

Branch of study: Intelligent Systems

Title: **Recurrent Neural Networks with Elastic Time Context in Language Modeling**

Category: Speech and Natural Language Processing

Instructions for project work:

1. Get familiar with language modeling with recurrent neural networks (RNN)
2. Get familiar with ways of learning longer context dependencies in RNNs
3. Implement some of these techniques using suitable machine learning tools
4. Compare these techniques on a standard data-set.
5. Suggest and implement way(s) to improve a selected technique by standard machine learning techniques, for example regularization.
6. Study the improvements in terms of complexity and data requirements.
7. Create a poster and/or video presenting your work.

Basic references:

- based on supervisor's recommendation

Requirements for the semestral defense:

Items 1 to 4

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

The Master's Thesis must define its purpose, describe a current state of the art, introduce the theoretical and technical background relevant to the problems solved, and specify what parts have been used from earlier projects or have been taken over from other sources.

Each student will hand-in printed as well as electronic versions of the technical report, an electronic version of the complete program documentation, program source files, and a functional hardware prototype sample if desired. The information in electronic form will be stored on a standard non-rewritable medium (CD-R, DVD-R, etc.) in formats common at the FIT. In order to allow regular handling, the medium will be securely attached to the printed report.

Supervisor: **Hannemann Mirko, Dipl.-Ing.**, DCGM FIT BUT

Beginning of work: November 1, 2015

Date of delivery: May 25, 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

Jan Černocký
Associate Professor and Head of Department

Abstract

This thesis describes an experimental work in the field of statistical language modeling with recurrent neural networks (RNNs). A thorough literature survey on the topic is given, followed by a description of algorithms used for training the respective models. Most of the techniques have been implemented using Theano toolkit. Extensive experiments have been carried out with the Simple Recurrent Network (SRN), which revealed some previously unpublished findings. The best published result has not been replicated in case of static evaluation. In the case of dynamic evaluation, the best published result was outperformed by 1 %. Then, experiments with the Structurally Constrained Recurrent Network have been conducted, but the performance could not be improved over the SRN baseline. Finally, a novel enhancement of the SRN was proposed, leading to a Randomly Sparse RNN (RS-RNN) architecture. This enhancement is based on applying a fixed binary mask on the recurrent connections, thus forcing some recurrent weights to zero. It is empirically confirmed, that RS-RNN models learn the training corpus better and a combination of RS-RNN models achieved a 30 % bigger gain on test data than a combination of dense SRN models of same size.

Abstrakt

Tato zpráva popisuje experimentální práci na statistické jazykovém modelování pomocí rekurentních neuronových sítí (RNN). Je zde předložen důkladný přehled dosud publikovaných prací, následovaný popisem algoritmů pro trénování příslušných modelů. Většina z popsaných technik byla implementována ve vlastním nástroji, založeném na knihovně Theano. Byla provedena rozsáhlá sada experimentů s modelem Jednoduché rekurentní sítě (SRN), která odhalila některé jejich dosud nepublikované vlastnosti. Při statické evaluaci modelu byly dosažené výsledky relativně cca. o 2.7 % horší, než nejlepší publikované výsledky. V případě dynamické evaluace však bylo dosaženo relativního zlepšení o 1 %. Dále bylo experimentováno i s modelem Strukturně omezené rekurentní sítě, ale ten se nepodařilo natrénovat k předpokládáným výkonům. Konečně bylo navrženo rozšíření SRN, pojmenované Náhodně prořidlá rekurentní neuronová síť. Experimentálně bylo potvrzeno, že RS-RNN dosahuje lepších výsledků v učení vlastního trénovacího korpusu a kombinace několika RS-RNN modelů přináší o 30 % větší zlepšení než kombinace stejného počtu SRN.

Keywords

Statistical Language Modeling, Recurrent Neural Network, Random Sparsity of Weights, Word History Representation

Klíčová slova

Statistické jazykové modelování, rekurentní neuronové sítě, náhodná řidkost vah reprezentace slovní historie

Reference

BENEŠ, Karel. *Recurrent Neural Networks with Elastic Time Context in Language Modeling.* Brno, 2016. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Hannemann Mirko.

Recurrent Neural Networks with Elastic Time Context in Language Modeling

Declaration

Hereby I declare that this diploma thesis describes an original work of the the author, under the supervision of Mirko Hannemann. All the relevant information sources used during preparation of the thesis are appropriately cited and included in the list of references.

.....
Karel Beneš
May 25, 2016

Acknowledgements

I would like to give my thanks to Mirko Hannemann, for his guidance, patience and a deep questions that did not dare to ask. I would also like to thank my fiancée for her endless support.

Contents

Chapter 1

Introduction

Since the early era of continuous speech recognition with large vocabulary, it has been crucial to compare different hypotheses by their adherence to the language of the utterance. For several decades, simple models based on the Markov assumption were defining the state of the art in the field.

Since the second decade of the third millennium, these models are surpassed by various forms of recurrent neural networks. After their initial success, recurrent neural networks have flourished in acoustic modeling, image captioning and machine translation. It is argued that the main advantage of such models is the compact representation of arbitrary long histories.

This project examines various aspects of the basic recurrent neural language model. Then, an enhancement of this model is proposed and its performance is assessed.

The work is based on a literature survey presented in Chapter 2. There, the neural networks used in language modeling are described, including different techniques used for tackling long-term dependencies. In Chapter 3, the process of training a neural network is described, with emphasis on techniques relevant mainly for the language modeling. The literature survey serves as basis for the experiments in Chapter 4 and inspiration for original proposed technique described in Chapter 5.

Experiments are conducted on a widely used dataset and all hyperparameters are explored. There is certain drawback of the chosen implementation, which is discussed in depth. Nevertheless, several interesting findings are presented which can not be found in the literature on the topic. Finally, it is shown how the drawback may be overcome and state of the arts results reached.

The proposed enhancement of the model is based on keeping only a subset of recurrent weights and is similar to the dropout technique. It is shown that using this technique allows for more effective training and such models combine better than the baseline.

Chapter 2

Neural Networks for Language Modeling

There have been some experiments with neural networks (NNs) for language modeling as early as of 1991 [35]. However, there has been more interest since 2003 [5]. Although the initial work in the area was done with traditional feedforward neural networks (FFNNs), a boom of recurrent neural networks (RNNs) has been observed in recent years. In this thesis, only a few RNN models are considered, for a wider overview refer to publications [41] or [33].

The task of language modeling is understood as defining a probability distribution over possible successor events given the history. Assuming that the universe of events is discrete with K distinct events, we can express the maximum likelihood (ML) approach of learning the parameters od the model by using the cross-entropy function (2.1) as the error function. In the context of language modeling, we refer to an event as word or token. Let us evaluate the model on a sequence of N tokens, with every token n being encoded as 1-of-K vector t_n and the response of network is denoted as vector y_n at every timestep. Then the cross-entropy is defined as a function of model parameters Φ and is computed as a sum of negative log-probabilities of these events, that would have been the correct predictions.

$$E(\Phi) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk} \ln y_{nk} \tag{2.1}$$

Throughout this chapter, several different issues are addressed: First, the task of language modeling is defined more precisely and traditional approaches to language modeling are presented. Next, FFNNs are introduced and general properties of NNs are discussed. In following sections, three recurrent models are defined. General properties of RNNs are discussed in the scope of the simplest one.

2.1 The Task of Statistical Language Modeling

Statistical language modeling is the task of estimating a probability distribution over some dictionary given a history w_1^{t-1} consisting of the previous words (2.2). For practical reasons, we can also define the probability of the whole sequence (2.3).

$$p(w|w_1^{t-1}) = p(w_t|w_1 w_2 \dots w_{t-1}) \tag{2.2}$$

$$p(s) = p(w_1|w_1^0) \cdot p(w_2|w_1^1) \cdot p(w_3|w_1^2) \cdot \ldots = \prod_i p(w_i|w_1^{i-1}) \tag{2.3}$$

For any but the simplest cases, the value of this probability is very low, possibly introducing an underflow when computed on an ordinary floating point architecture. Also, the length of a sequence may vary significantly from a few words in a single sentence to thousands of words in case of processing a whole document. Thus, it is customary to report per-word statistics. Following Equation (2.3), we compute these statistics as a geometric mean over the whole sentence. Finally, it is usual in machine learning to work with error, which is the inverted value of the target function.

So we get the *perplexity* (2.4) measure. Typical values of perplexity range from 120 to 160 on unseen data, depending on complexity of the task and power of the model applied. This number is equivalent to the average number of words the model would predict as possible, given that it would only give a uniform probability to the subset of possible words in the vocabulary and zero to the other words.

$$\text{PPL} = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{p(w_i|w_1^{i-1})}} \tag{2.4}$$

Taking logarithm of the formula for computing perplexity, we receive an error measure typically referred to as *per-word entropy* (2.5). It is an average value of cross-entropy between the model and some test data. The value of per word entropy is binary logarithm of perplexity, thus typical values range from 6.9 to 7.3.

$$\text{per-word entropy} = -\frac{1}{N} \sum_{i=1}^{N} \log_2 p(w_i|w_1^{i-1}) \tag{2.5}$$

There has been some opposition to statistical language modeling as such, most notably from Noam Chomsky, 1969:

> *But it must be recognized that the notion of „probability of a sentence“ is an entirely useless one, under any known interpretation of this term.*

Nevertheless, the probability of a sentence is utilized in practical applications such as speech recognition, machine translation and data compression.

For many years, state of the art in language modeling was defined by *n-gram* models. An n-gram is an n-tuple of consecutive words. These models exploit the Markov assumption on the data, modeling the probability of the i-th word conditioned only on the $n-1$ previous words (2.6).

$$P(w_t|w_1 w_2 \ldots w_{t-1}) = P(w_t|w_{t-n+1} w_{t-n+2} \ldots w_{t-1}) \tag{2.6}$$

Since there is a limited number of possible $n-1$ long histories, training of an n-gram model is constituted by counting occurrences of the given n-grams in a large training corpus. However, not all n-grams are seen in the training data, therefore different smoothing methods are used in order to assign non-zero probability to such unseen n-grams.

It is possible to represent an n-gram model as weighted a finite automaton by representing each history as a unique state. The WFST approach allows to efficiently find the most likely sequence of states and thus words. This can be further exploited in the task of speech

recognition: Other sources of information – the pronunciation of words as well as acoustic properties of the signal itself – can be encoded as WFST, allowing a compact representation of the whole problem [34].

Over the years, several improvements of n-grams model have been proposed, that try to overcome different weaknesses of the standard model. For instance cache models [32] [28] address the ability to access history beyond the $i - n + 1$th word. Clustering (or classing) models [21], define *categories*, such as *day-of-week* or *color*. All words of a single category are treated as instances of the same token during counting of n-grams. Therefore, we receive a more robust estimate of the n-gram probability of these tokens.

2.2 Feed-Forward Networks

A neural network (NN) is a general connectionist model, which consists of a set of artificial neurons. Every artificial neuron computes a scalar function of its multidimensional input. Inputs of neurons are determined by the interconnection of neurons. Usually, most of neurons compute the same type of function, exception being the neurons in the last layer. Inspired by the function of a biological neuron, the function of an artificial neuron can be expressed as a composition of a basis function $g(\cdot)$ and an activation function $f(\cdot)$.

$$y = f(g(\mathbf{x})) \tag{2.7}$$

Although there are other possible basis functions, such as the radial basis function [12], the linear basis function $g(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$ is dominant in todays NNs.

Activation functions are nonlinear and can be either continuous or step-functions. A discussion of several prevalent activations is given in Subsection 2.2.1. Regardless of the specific nature of a given nonlinearity, the presence of a non-linearity is crucial for the overall function of the network: Since linear transformations are closed under composition, a NN without nonlinearities would be restricted to linear properties of the input.

A FFNN is a learnable model defining a function $q : X \to Y$, which consists of several separate layers as demonstrated in figure 2.1. Since every neuron j in any given layer performs the same operation $y = f(\mathbf{w}_j^T\mathbf{x}_j)$, we can consider the whole layer as a vector of these neurons, writing $\mathbf{h}_i$ for hidden layer i. Then, we can express the computation of all neurons in a single layer as $\mathbf{h}_{i+1} = f(\mathbf{W}\mathbf{h}_i)$, where the activation function $f(\cdot)$ is applied element-wise.

2.2.1 Nonlinearities Used in Artificial Neurons

Following the original argument from the 1950's [30], it has been proven [14], that with a *sigmoidal* activation function (2.8), a FFNN can approximate any continuous function to arbitrary precision, given the hidden layer is large enough. This has been later generalized [24] to any activation function, that is continuous, bounded and nonconstant. It is noteworthy, that the early works were motivated by Hilbert's 13th problem, which asks, whether a solution of a seventh-degree equation can be expressed as a composition of finite set of two-parameter functions.

$$\sigma(x) = \begin{cases} 1 & \text{as } x \to +\infty \\ 0 & \text{as } x \to -\infty \end{cases} \tag{2.8}$$

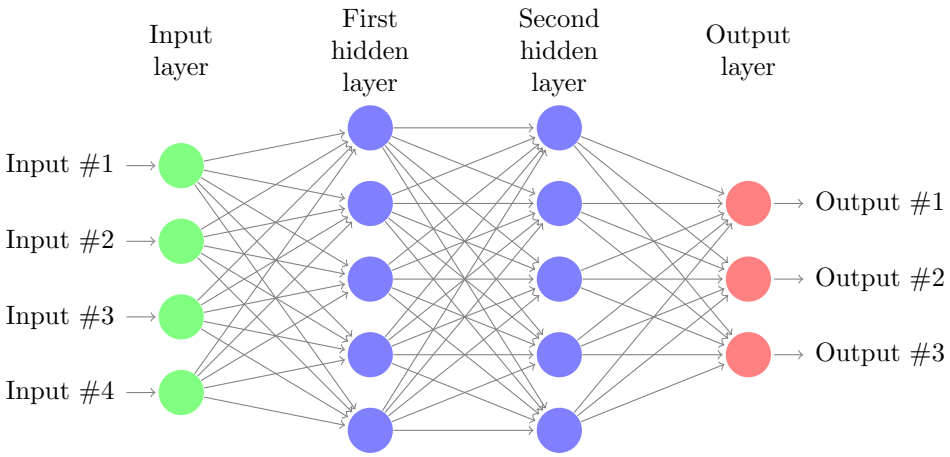


Figure 2.1: General schema of feedforward neural network (FFNN). Each node represents a single neuron. Each arrow represents a synaptic connection and weight is associated with it. Nonlinear activation of the neurons is not shown.

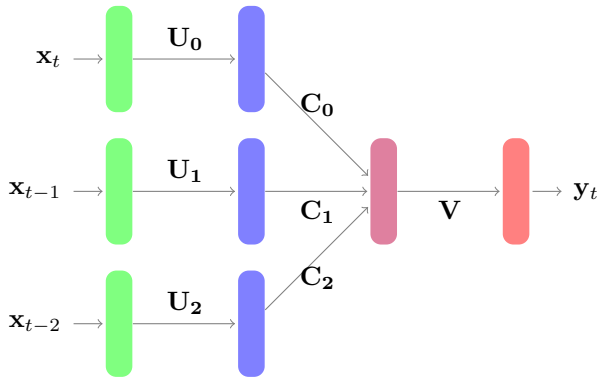
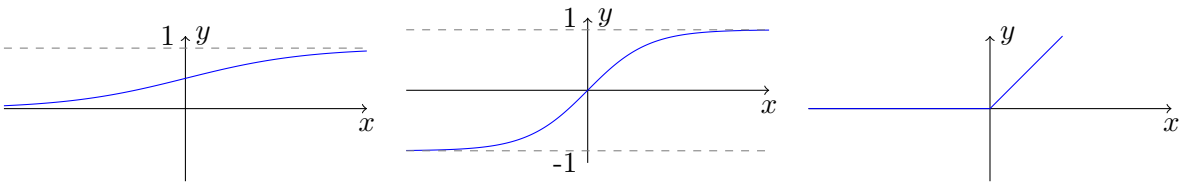


Figure 2.2: A feedforward neural network (FFNN) used as a language model. It takes a finite number of words as input, three in this case. Every word i has its own, independent projection matrix U_i, depending on the position it is at. The pre-last hidden layer, called *compression layer* (purple), is used to pull the most important information from the word vector. This allows the output layer to concentrate on the actual prediction of the output word.



(a) Logistic sigmoid (b) Hyperbolic tangent (c) Rectified linear unit

Figure 2.3: Different nonlinearities used as activation functions. In this work, the logistic sigmoid is used. Note that hyperbolic tangent is is the same function, just rescaled. Refer to Figure 3.1 for derivatives of these functions.

There are three dominant activation functions used today. The logistic sigmoid (Fig. 3.1a), as given by (2.9), arises naturally from logistic regression for binary classification [10]. However, it has been shown [18], that hyperbolic tangent(Fig. 3.1b), as given by (2.10), allows faster training and a NN using it will typically reach a better local optimum.

Finally, the rectified linear unit (ReLU)(Fig. 3.1c) has been recently proposed [19] to allow a deeper propagation of gradients[1]. The ReLU activation is also more biologicaly plausible and leads to a sparse representation of the input in the hidden layers.

$$\sigma(a) = \frac{1}{1+e^{-a}} \quad (2.9) \qquad \tanh(a) = \frac{e^{a}+e^{-a}}{e^{a}-e^{-a}} \quad (2.10) \qquad ReLU(a) = \begin{cases} a & a > 0 \\ 0 & \text{otherwise} \end{cases} \quad (2.11)$$

The output layer neurons are treated differently, because we want the output of the network to define a probability distribution, i.e. we want all the outputs to be positive and to sum up to one. For this purpose, the softmax function is used. It is in principle a normalized per-element exponentiation of the basis function. Since the underlaying function is exponential, much greater output value is assigned to neurons with just slightly greater basis value. Thus the name „softmax“, since the function tends to assign almost 1 to the neuron with the highest basis value and almost zero to the rest of them.

$$\text{softmax}\left(\begin{bmatrix} o_1 \\ o_2 \\ \vdots \\ o_K \end{bmatrix}\right) = \begin{bmatrix} \frac{e^{o}_{1}}{Z} \\ \frac{e^{o}_{2}}{Z} \\ \vdots \\ \frac{e^{o}_{K}}{Z} \end{bmatrix}, Z = \sum_{k=1}^{K} e^{o_k} \quad (2.12)$$

2.2.2 Neural Networks with several Hidden Layers

Although one hidden layer can be proven to provide universal capabilities to a NN, it is computationally beneficial to use more successive hidden layers. Advantages of increased depth have been argued [4][7], proven [25] and shown to take part for instance in the human vision [31] system. The principal advantage of a deeper NN is the ability to use an output of

[1] The necessity, effects and problems of gradients propagation are discussed in Chapter 3.

a given neuron as the input for all neurons in the following layer, thus relieving them from the necessity to compute the associated feature themselves. Stacking more layers therefore allows for transforming the input into features of increasing level of abstraction.

Until 2006, there was little success in training neural networks with more hidden layers in a supervised manner. This was mainly due to the highly non-convex objective function and difficulties in expressing the influence of parameters close to the input layer – as the output of these neurons is used in a complex way by large parts of the network, it is difficult to find out figure out a change of its parameters that would improve the overall behaviour of the network.

In 2006, a pretraining method was published [22], which allows training layer-wise in an unsupervised manner, which is followed by supervised fine-tuning of the weights in the whole network.

There were several attempts to remove the unsupervised pretraining from the procedure. Supervised training with adding layers one at a time [6] did show improvement over whole-network-at-once approach but did not reach performance of unsupervised pretraining. Finally, it has been shown in 2010 [19], that using the rectifiers as activations, very deep networks can be trained directly in a supervised manner.

2.2.3 Regularizing Techniques for Improving Generalization Properties of Neural Networks

Since NNs used as probabilistic models are discriminative [2] and we train them using the ML approach, some techniques have to be applied to assure that overfitting to the training data does not occur.

A basic technique to avoid overfitting is to reduce the dimensionality of the parameter space. In the context of NNs, this equals to using smaller hidden layers. Although there are some sophisticated methods such as adding hidden units during the learning [16], it is typically up to the designer of the network to choose reasonable sizes of individual hidden layers. Thus, they are usually picked by intuition and performance of different sizes is compared on a held-out validation set. The effect of the hidden layer size is discussed in the experiments sections.

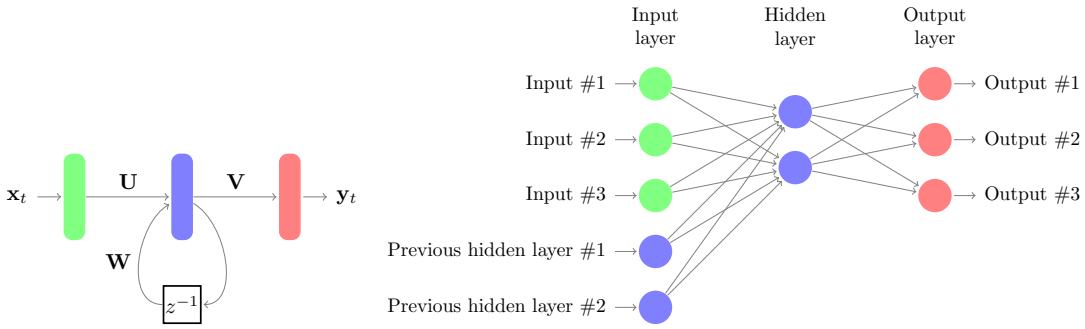
Another option is to suppress large parameter values. This is generally done by enhancing the loss function by a penalty term (2.13) [10], where $||\cdot||$ denotes the norm of the parameters. A square of the L_2 norm and the L_1 norm are typically used as penalty functions.

$$loss(\Phi) = E(\Phi) + \beta||\Phi|| \tag{2.13}$$

For deep neural networks (DNNs), a very successful method of *dropout* was proposed recently [45]. This method introduces a nonzero probability, that an input of a neuron will be replaced by zero, effectively turning the neuron off for a given training sample. This improves the ability of the network to generalize, because following neurons can not overly rely on the output of any given neuron, and also improves learning because it help to break symmetries between neurons. This effect will be further discussed in Chapter 3.

Nevertheless, it is uncommon for today's NNs used as language models (LMs) to overtrain. The reason is, that the amounts of data (millions to billions of tokens) greatly surpass

[2] They estimate the conditional probability $p(t|x)$ of some target t given input x, as opposed to generative models estimating probability $p(x,t)$ of the data itself.



(a) Packed diagram of RNN. The z^{-1} block denotes a delay for single timestep.

(b) Detailed diagram of RNN operation in a single timestep. Values of the neurons in the hidden layer are concatenated to the input in the next step.

Figure 2.4: General schema of an recurrent neural network (RNN). This structure is also the one proposed by early work of Elman [15], known as simple recurrent network (SRN). An RNN expects the input to be a sequence of single elements. The length of the sequence is unlimited and the model keeps a compressed representation of all the previous samples in its hidden layer.

the memorization abilities of any reasonably sized NNs. Thus, the main issue with training NN LMs is underfitting.

2.3 Fully Connected Recurrent Neural Networks

A Recurrent neural network (RNN) is such a NN, that uses its hidden state from processing the last sample as an additional input for processing the current sample. The first note of such RNN is found in the work of Elman [15].

The first successful [36] RNN model used for language modeling was a RNN using hidden units with logistic sigmoid activation (2.9). The structure of the model is directly following the general Figure 2.4. A single processing step (consuming one input and producing one output) is thus defined as follows: The output is given by Equation (2.15) and the hidden state is given by Equation (2.14).

$$\mathbf{h}_t = \sigma(\mathbf{U}\mathbf{x}_t + \mathbf{W}\mathbf{h}_{t-1}) \tag{2.14}$$

$$\mathbf{y}_t = \text{softmax}(\mathbf{V}\mathbf{h}_t) \tag{2.15}$$

The matrix $\mathbf{U}$ captures the input mapping. It is essentially equivalent to the weight matrix of a single layer of an FFNN, but given that the input encodes discrete events (words) in the 1-of-K encoding scheme, every single column of $\mathbf{U}$ can be understood as a word vector of the respective word. Therefore, the weight matrix $\mathbf{U}$ introduces a continuous vector space representation of discrete events.

The idea of word vectors has flourished since their implicit introduction, and word vectors are used separately now, as an input to natural language processings (NLPs) applications. A detailed discussion on definition, computation and properties of word vectors can be found in recent papers [38] [39] [43].

The output layer consisting of weight matrix $\mathbf{V}$ and the softmax output activation is a standard multi-class logistic regression, as used in any NN for multi-class classification.

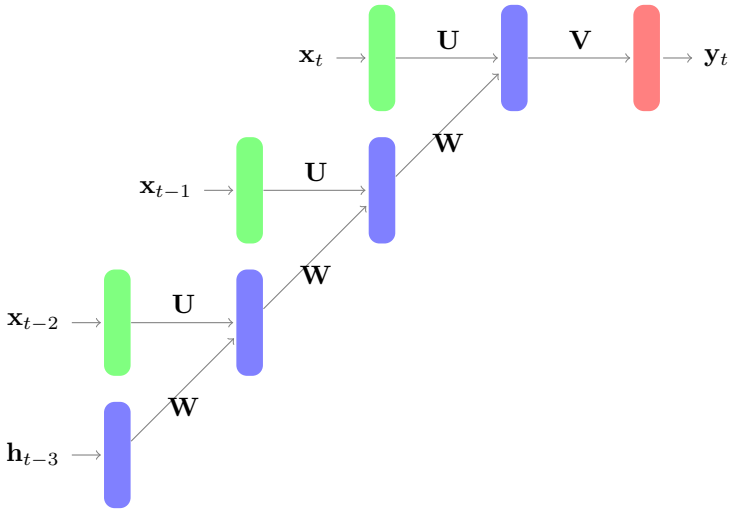


Figure 2.5: A recurrent neural network (RNN) expanded for three timesteps. This diagram shows that an RNN can be interpreted as a deep neural network (DNN) with parameters shared across the layers. Note that every box represents whole layer of NN, thus consists of hundreds to thousands of neurons.

The matrix $\mathbf{W}$ represents the recurrent weights. The state $\mathbf{h}_{t-1}$ of the hidden layer is multiplied by these weights and then added to the input of hidden units at the step t. The actual value of this weight matrix is crucial for the overall behaviour of the network: Every input x_t has in general some influence on every hidden state $h_{t'}, t' \geq t$. In every timestep t, this influence is multiplied by $\mathbf{W}$, so it can either gradually diminish, which is typically more desirable, or suppress influence of following words. It can not actually grow, because the definite upper bound is saturating all the neurons in the hidden layer, regardless of the input.

As emphasised in Figure 2.5, an RNN may be understood as a DNN with output spread in time. Note that this DNN has parameters shared throughout the whole depth. Therefore, problems similar to what is experienced in training DNNs may arise in training RNNs.

Given that hidden layer $\mathbf{h}_t$ is the only input to the output layer, which actually computes the probability $p(w_t|h)$, it is clear that the hidden layer provides compact history representation in the form of a vector in a continuous n-dimensional space. The continuous representation is the key aspect for the improved performance [36], as it is a way to avoid the curse of dimensionality experienced by n-gram models.

Given that H denotes dimensionality of the hidden layer and V size of the vocabulary, the total number of learnable parameters is equal to $V \times H + H^2 + (H + 1) \times V$. Only the H^2 term is inherent to the type of recurrent connection used, as the terms $V \times H$ and $(H + 1) \times V$ are directed by the vocabulary size and are inevitable in any NN-based statistical language model.

The output layer is a computational bottleneck. Not only it involves the biggest matrix multiplication, but also $|V|$ exponentiations. Therefore, a factorization method is often used to obtain a speed-up: Words are clustered into *classes*, every word w receiving its class c_w, and a *hierarchical softmax* (2.16) is than applied to predict the next word. With the number of classes close to $\sqrt{|V|}$, the complexity drops from $\mathcal{O}(HV)$ to $\mathcal{O}(H\sqrt{V})$. In practice, the speed-up ratio of 20 is not unusual, at a slight decrease of the performance.

Origins of the hierarchical softmax are traced back to Goodman's work [20].

$$p(w_{t+1}|w_1^t) = p(w_{t+1}|c_{w_{t+1}}, w_1^t) \cdot p(c_{w_{t+1}}|w_1^t) \tag{2.16}$$

2.4 Long Short-Term Memory Model

It has been shown [8] and recently discussed [42], that despite their general potential, simple RNNs are prone to many problems preventing them from learning long-term dependencies. A brief discussion of these problems will be given in Chapter 3.

As a model capable of learning long-term dependencies, the long short-term memory (LSTM) model [23] was introduced in the late 1990s. In this model, a simple neuron in the recurrent layer is replaced by a *memory cell*, which is able to store information for arbitrary long timespans.

An LSTM is typically understood just as a single layer of a possibly deeper architecture. To get the same view of a simple RNN, one would consider it as defined only by Equation (2.14), allowing the input to be computed by some previous layers.

Denoting the element-wise product (Hadamard product) by $\odot$, we can express the output $\mathbf{h}_t$ of an LSTM layer at timestep t, given input $\mathbf{x}_t$:

$$\mathbf{p}_t = \tanh(\mathbf{U}\mathbf{x}_t + \mathbf{W}\mathbf{h}_{t-1} + \mathbf{b}_u) \odot \sigma(\mathbf{I}_x\mathbf{x}_t + \mathbf{I}_h\mathbf{h}_{t-1} + \mathbf{b}_i) \tag{2.17}$$

$$\mathbf{m}_t = \mathbf{m}_{t-1} \odot \sigma(\mathbf{F}_x\mathbf{x}_t + \mathbf{F}_h\mathbf{h}_{t-1} + \mathbf{b}_f) + \mathbf{p}_t \tag{2.18}$$

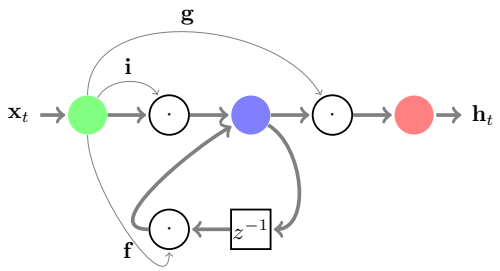
$$\mathbf{h}_t = \tanh(\mathbf{m}_t) \odot \sigma(\mathbf{G}_x\mathbf{x}_t + \mathbf{G}_h\mathbf{h}_{t-1} + \mathbf{b}_g) \tag{2.19}$$

In these equations, $\mathbf{m}_t$ denotes internal memories of the LSTM and $\mathbf{p}_t$ denotes already gated input. The weight matrices I_x, F_x, G_x are all of the same size and transform the input into *gating signals* (explained below), similarly the matrices I_h, F_h, G_h are of the same size and operate on the output of the LSTM layer from the previous timestep. The terms b_u, b_i, b_f, b_g are biases in respective transformations. Note, that the output $\mathbf{h}_t$ is typically hidden within the whole network, it is the LSTM analogy of $\mathbf{h}_t$ in the simple RNN. Thus its linear transformations are denoted by $\mathbf{A}_h$. Operation of the LSTM is illustrated in Figure 2.6.

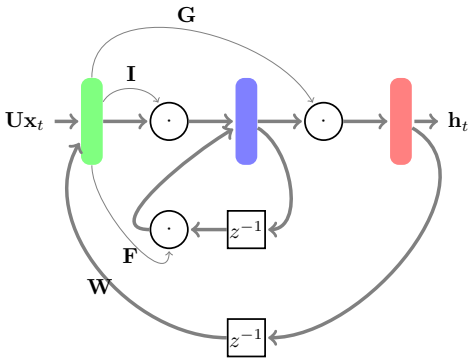
Every element-wise product has an interpretation of *gating* with strong influence on operation of the LSTM: Gating in (2.17) protects the internal memory of LSTM cells from input, gating in (2.18) allows to forget current memories and finally in (2.19), parts of the memories can be blocked from spoiling the output. As the gating is done by element-wise multiplication, the LSTM can learn to gate specific parts of the signal from passing further, making it a very flexible model.

This flexibility comes with a cost: Having the actual input plus three gates, taking full information from input, the model has roughly 4 times as many parameters as a simple RNN. The time needed for the computation of a single timestep is also greater than in a simple RNN. Nevertheless, this is not so important, as the most computationally expensive operation is usually the output layer which is common for both models.

Inspired by the LSTM architecture, gated recurrent unit (GRU) model was proposed [13], achieving similar results with only one gate per unit. In a recent exploration paper by Józefowicz et al. have performed an exhaustive search of architectures similar to the LSTM and



(a) Detail of a single LSTM cell. Note that the main difference with respect to a simple neuron of a regular RNN is the additional recurrent connection within. Letters *i,f,g* denote the input, forgetting and output gating respectively.



(b) One layer of LSTM. Note that the gating input to respective gates is computed from the whole input $\mathbf{x}_t, \mathbf{h}$. Capital letters denote linear transformation by respective matrices.

Figure 2.6: Diagram of long short-term memory (LSTM) operation. Thick lines represent flow of actual data, while thin ones are so called *gating* inputs, that open or close respective *gates* by point-wise multiplication. Green unit represents input, red output and blue is the internal memory. In both diagrams, both nonlinearities and biases are omitted.

GRU. However, they did not find any, which would outperform both LSTM and GRU on all of the considered tasks [29].

2.5 Structurally Constrained Recurrent Neural Networks

The so called Structurally Constrained Recurrent Network (SCRN) model was proposed by Mikolov et al. [37] as an alternative to the complex gating systems. This model enhances the simple recurrent network (SRN) by adding dedicated neurons, that are constrained as to store information about longer history. Several attempts to achieve similar results using similar techniques were reported, e.g. [40] [27] or [3]. This section describes the Structurally Constrained Recurrent Network (SCRN) model.

With the dedicated neurons added, a single computational step of a SCRN is defined by the following equations:

$$\mathbf{s}_t = (1-\alpha)\mathbf{B}\mathbf{x}_t + \alpha\mathbf{s}_{t-1} \tag{2.20}$$

$$\mathbf{h}_t = \sigma(\mathbf{U}\mathbf{x}_t + \mathbf{W}\mathbf{h}_{t-1} + \mathbf{P}\mathbf{s}_t) \tag{2.21}$$

$$\mathbf{y}_t = \text{softmax}(\mathbf{V}_h\mathbf{h}_t + \mathbf{V}_s\mathbf{s}_t) \tag{2.22}$$

The hidden state vector $\mathbf{h}$ and the output vector $\mathbf{y}$ have the same meaning as in SRN defined by Equations (2.15) and (2.14). The vector $\mathbf{s}$ captures the state of the longer memory neurons. These neurons are not affected by the regular hidden neurons, which makes the changes in $\mathbf{s}$ much slower. Thus, these neurons keep information for a longer time, making it possible for the network to learn from longer-term dependencies in the text.

Notice that no nonlinearity is applied as activation of these neurons, resulting in more effective back-propagation of gradients, as discussed in detail in Section 3.2. Also this makes

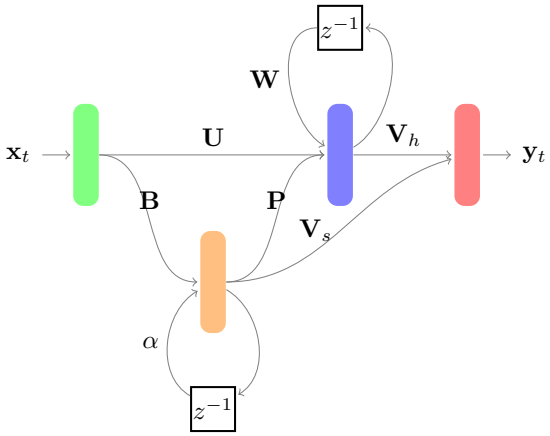


Figure 2.7: Schema of Structurally Constrained Recurrent Network (SCRN). It is an enhanced model of simple recurrent network (SRN) from Figure 2.4a. Note that the values of longer memory neurons (orange) from last timestep are not transformed by any linear transformation but element-wise scaling. Complete operation of the SRN is given by Equations (2.20), (2.21) and (2.22).

them effectively accumulate their input. Therefore, slow neurons represent an exponentialy decaying bag-of-words feature.

Although the decay parameter α could be learnt and even set to different values for separate neurons, the authors do not advert it. They argue and show by experiments, that with sufficient number of regular hidden neurons [3], the choice of α is arbitrary as long as it is close to 1.

The pair of hidden vectors can also be viewed as a single hidden vector $\mathbf{c}$, where $\mathbf{h}$ and $\mathbf{s}$ are concatenated. The computation of new value of the hidden vector can be expressed as follows:

$$\mathbf{c}_t = \text{nonlinearity}\left(\left[\begin{array}{c|c} W & P \\ \hline 0 & \alpha I \end{array}\right] \mathbf{c}_{t-1} + \text{input projection}\right)$$

We can see that nearly whole lower part of the united recurrent matrix is forced to zero value, hence the name *structurally constrained* recurrent network.

[3] Depending on the dataset, as few as 100 can be considered sufficient.

Chapter 3

Algorithms for Model Implementation and Training

The most prevalent methods for training NNs are based on gradient descent methods. That is, the error estimate $\hat{\mathbf{E}}(\Phi_t)$ is computed for the current setting Φ_t of parameters and so are its first derivatives with respect to all parameters ϕ_i.

Given that we have computed the current gradient of the error with respect to every parameter, there are several training procedures used. The basic approach, *batch training*, accumulates the errors from all training samples, then computes the gradient and updates the parameters. This method is solid in the sense that it uses an estimate of error the error as accurate as possible. However, it is not used in practice due to slow convergence and being prone to ending in a shallow local optimum.

To overcome this issue, *stochastic gradient descent* can be applied. In this training schema, the weights are updated after every sample. Stochastic gradient descent typically converges much faster and the noise in the estimate of the error allows it to escape from most of the local optima. However, the computation of gradients is a costly operation and computing it after every sample may be too expensive. Therefore it is common to compute the weight update after a fixed number of samples. These *minibatches* can be picked so that there is an example of each output class in each batch, so that the network has to learn to discriminate between them.

3.1 Computing Gradients with Backpropagation

The essential operation necessary for this kind of optimization is the computation of the gradients. As will be shown, it is possible to compute the gradients with respect to all parameters of a neural network in an efficient manner. This method is called backpropagation (back propagation of errors), and it can be derived as follows:

Let us have a FFNN with several hidden layers and a softmax as the output layer. The hidden layers are assumed to have a logistic sigmoid as their activation function. The discussion of other nonlinearities defined in 2.2.1 with respect to the backpropagation is given Subsection 3.1.1. Then every output y_i is computed as:

$$y_i = \frac{e^{o_i}}{\sum_j e^{o_j}} \tag{3.1}$$

Putting regularization aside for the moment, we optimize the cross-entropy only, so we

minimize the function:

$$E = -\sum_k t_k \log y_k = -\sum_k t_k \frac{e^{o_k}}{\sum_j e^{o_j}} \tag{3.2}$$

Given that last hidden layer of network is $\mathbf{h}_M$, the o_is are computed as:

$$\mathbf{o} = \mathbf{V}\mathbf{h}_M + \mathbf{b}_v \tag{3.3}$$

So let us assume that we have processed a single training sample $\mathbf{x}$ and received an output $\mathbf{y}$ while we know the correct output ought to be $\mathbf{t}$. Using the chain rule of derivation, we first take the partial derivative of the error with respect to the weight matrix $\mathbf{V}$:

$$\frac{\partial E}{\partial \mathbf{V}} = \frac{\partial E}{\partial \mathbf{o}} \frac{\partial \mathbf{o}}{\partial \mathbf{V}} = (\mathbf{t} - \mathbf{y}) \frac{\partial [\mathbf{V}\mathbf{h}_M + \mathbf{b}_v]}{\partial \mathbf{V}} = (\mathbf{t} - \mathbf{y})\mathbf{h}_M^T \tag{3.4}$$

Similarly, partial derivative can be taken with respect to the biases of the output:

$$\frac{\partial E}{\partial \mathbf{b}_v} = \frac{\partial E}{\partial \mathbf{o}} \frac{\partial \mathbf{o}}{\partial \mathbf{b}_v} = (\mathbf{t} - \mathbf{y}) \frac{\partial [\mathbf{V}\mathbf{h}_M + \mathbf{b}_v]}{\partial \mathbf{b}} = (\mathbf{t} - \mathbf{y}) \tag{3.5}$$

Finally, we move deeper to the network and investigate influence of the pre-last hidden layer $\mathbf{h}_{M-1}$ on the result. We know that the last hidden layer $\mathbf{h}_M$ was computed from $\mathbf{h}_{M-1}$ as follows:

$$\mathbf{h}_M = \sigma(\mathbf{W}_M \mathbf{h}_{M-1} + \mathbf{b}_M) = \sigma(\mathbf{o}_M) \tag{3.6}$$

So continuing with the chain rule of derivation, we get:

$$\frac{\partial E}{\partial \mathbf{W}_M} = \frac{\partial E}{\partial \mathbf{o}} \cdot \frac{\partial \mathbf{o}}{\partial \mathbf{h}_M} \cdot \frac{\partial \mathbf{h}_M}{\partial \mathbf{W_M}} = \frac{\partial E}{\partial \mathbf{o}} \cdot \frac{\partial \mathbf{o}}{\partial \mathbf{h}_M} \cdot \frac{\partial \mathbf{h}_M}{\partial \mathbf{o}_M} \cdot \frac{\partial \mathbf{o}_M}{\partial \mathbf{W_M}} = (\mathbf{t}-\mathbf{y}) \cdot \mathbf{V}^T \odot \mathbf{h}_M \odot (1-\mathbf{h}_M) \odot \mathbf{h}_{M-1}^T \tag{3.7}$$

Here, we have taken advantage of the specific form of the derivative of the logistic sigmoid, as defined in (3.10).

We could take the derivative with respect to the biases $\mathbf{b}_M$ analogically.

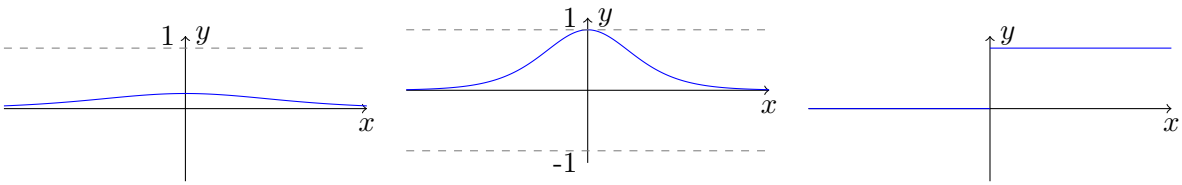
We can make get the key insight into the process of computing gradients by defining *errors* at the investigated layers:

$$\delta^v = \frac{\partial E}{\partial \mathbf{o}},\ \delta^M = \frac{\partial E}{\partial \mathbf{o}_M},\ \delta^{M-1} = \frac{\partial E}{\partial \mathbf{o}_{M-1}},\ \ldots \tag{3.8}$$

Finally, we the observe relation between errors at subsequent layers and the relation of errors at layers to errors of particular parameters. These relations follow from the assumed uniform feed-forward architecture of the network:

$$\delta^{m-1} = \delta^m \cdot \frac{\partial \delta^m}{\partial \mathbf{h}_m} \cdot \frac{\partial \mathbf{h}_m}{\partial \mathbf{o}_{m-1}} = \delta^m \cdot \mathbf{W}_m^T \odot \mathbf{o}_{m-1} \odot (1 - \mathbf{o}_{m-1}) \tag{3.9}$$

Continuing to deeper layers $\mathbf{h}_n, 1 \leq n < m$, we see that we always use the error computed with respect to the activity $\mathbf{o}_{n+1}$ of neurons in following layer. This allows to iteratively compute these partial errors for deeper layers and then taking the partial derivatives to the respective weights. The advantage over a naïve approach is in the usage of already precomputed errors associated with layers closer to the output.



(a) Logistic sigmoid (b) Hyperbolic tangent (c) Rectified linear unit

Figure 3.1: Derivatives of different nonlinearities used as activations in neural networks (NNs)

3.1.1 Derivatives of the Activation Functions

It is also useful to investigate the derivatives of nonlinear activation functions used in NNs.

Both logistic sigmoid and hyperbolic tangent have bell-shaped derivatives with maxima at zero, which leads to a decrease of the magnitude of errors in deeper layers. The derivative of hyperbolic tangent is sharper, which is one of the reasons for the higher speed of convergence. Given that the value of the nonlinearity has been computed as $\mathbf{h}$, it is computationally simple to get the derivative at the examined point for both logistic sigmoid (3.10) and hyperbolic tangent (3.10).

$$\sigma'(a) = \sigma(a)(1 - \sigma(a)) \tag{3.10}$$

$$\tanh'(a) = 1 - \tanh^2(a) \tag{3.11}$$

The ReLU activation has a constant derivative for any positive input value, which allows theoretically unlimited propagation of the gradients. This effect helps greatly for training DNNs and some initial work has been done in training RNNs as well [3]. Even though its derivative at zero is not defined, it is not a problem in practice, as both left derivative (0) and right derivative (1) can be used. Computing the derivative of ReLU at a given point is trivial (3.12).

$$\mathrm{ReLU}'(a) = \begin{cases} 1 & a > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.12}$$

It is suggested [18] to take the particular form of the nonlinearity into account when initializing the weights of the network. The initialization should be done in such a way, that most neurons are likely to have the activations in regions of non-zero derivative, which prevents early saturation of the nonlinearities.

3.2 Backpropagation through Time

The backpropagation of errors presented in Section 3.1 is defined for FFNNs. However, an extension to RNNs is straightforward and follows the unpacked diagram (as presented in Figure 2.5) of RNN operation. The schema of gradient flow in an RNN is shown in Figure 3.2.

The core of the backpropagation through time is captured by the equation for computing error vectors at different depths of the network (3.13). The formula has interesting implications which are further discussed.

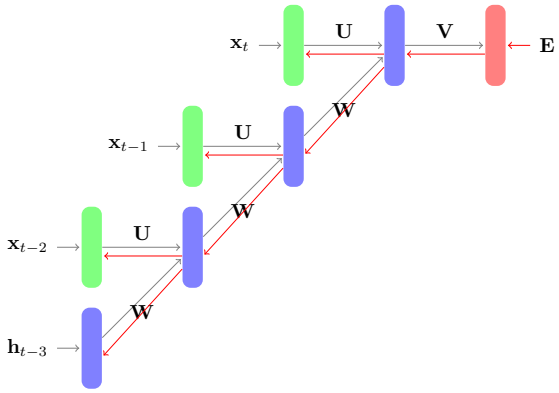


Figure 3.2: Schema of error backpropagation through time. Note that it is the usual backpropagation, just the network is an expanded RNN. Differences introduced by extensive weight sharing between the layers are discussed in the text.

$$\delta^D = \prod_{i=T-D}^{T} \left[(1 - \mathbf{h}_i) \odot \mathbf{h}_i \odot \mathbf{W}^T\right](\mathbf{t} - \mathbf{o}) \tag{3.13}$$

As the error is backpropagated through the same matrix over and over, it is actually multiplied by a power of the recurrent weights matrix. As follows from linear algebra, the power may happen to grow exponentially, resulting in so called *gradient explosion*. This effect has been thoroughly studied [42] and several solutions were proposed. Most of them consist of thresholding the gradient, either element-wise or as whole vector. Therefore, the exploding gradient is solved at the level of computing gradients, regardless of recurrent model used.

On the other hand, the gradual element-wise multiplication by the derivative of the activation introduces so called *gradient vanishing*. This effect is further increased if all eigenvalues of $\mathbf{W}$ are less than one. Therefore, it is difficult to learn longer temporal dependencies using this technique. In practice, prolonging the backpropagation over more than six time steps has been shown to have negligible effect [36].

Both the LSTM and SCRN try to overcome the vanishing gradient by enhancements to the model. The LSTM does so by introducing the inner memory loop, where the memories of the network are not transformed by any linear or non-linear transformation. The memories can only be suppressed by the forgetting gate, but the network can learn to keep the gate open. This is achieved by keeping the input to the respective gate high enough, as the logistic sigmoid saturates fast. The SCRN introduces slower neurons, which are not affected by any nonlinearity either. By setting a reasonably high α, the exponential decay of gradients is under control and the network can benefit from backpropagating over tens of words.

3.3 Overview of the Learning Process of a Recurrent Neural Language Model

Regardless of its structure, the recurrent neural network is trained on a dedicated training corpus. The basic strategy for learning the structure of the text is to process it word by word, as illustrated by Figure 3.3. This is the original strategy used in [36]. It is noteworthy

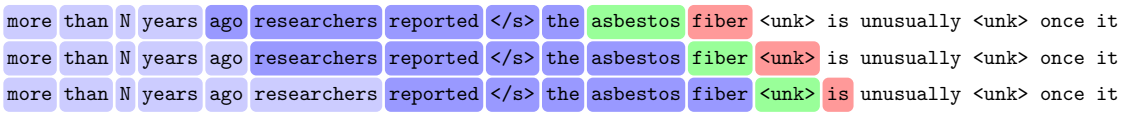


Figure 3.3: Learning the weights with backpropagation through time using a sliding window over the training text. The network processes one word at a time (green) and makes a single prediction (red word). Therefore, the weights are updated once per every word. Although the gradients are backpropagated only five steps into the history (dark blue words), previous words also influence the process, via the hidden state of the network.

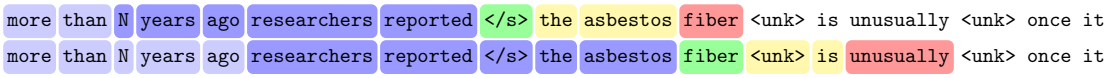


Figure 3.4: Updating the weights only once every 3 words. Green words are used for input only, red are only used as targets. The yellow words are used first as target and then as input. Backpropagation is done over five timesteps (words in dark blue), earlier words have only the effect of providing hidden state.

that this approach is typically implemented so that every single hidden vector is computed just once and only the current values of the parameters are stored. This implies that during the back-propagation, the gradients are computed with different recurrent weights than the forward pass of the respective hidden vectors. Nevertheless, it does not lead to any degradation in practice and experimental results suggests that it is even beneficial.

The backpropagation through time is a computationally costly operation. Therefore, it is often done only once per some segment of words, as illustrated in Figure 3.4. This is a sequential equivalent of the minibatch training used in the training of FFNNs. It trades the computational speed for the speed of convergence and it is not recommended to use a too long update period [36].

When training a FFNN for classification, it is customary to shuffle the training examples after each epoch. It is empirically observed, that it helps the stochastic gradient descent to avoid getting stuck at local optima. Also, it helps the network to learn generally valid facts about the data, rather than to approximate some false relations between successive samples. When training RNNs, it is necessary to keep the order of samples in a sequence, because the RNN is used for that very reason, to learn the dependencies between successive samples. However, shuffling may be done at a higher level, for instance shuffling sentences or blocks of sentences. When shuffling is done at the sentence level, the network learns as if sentences were independent, i.e. it cannot learn any memories which would help it at the beginning of a new sentence.

Finally, the network can be trained from multiple streams in parallel. This is a direct equivalent to the minibatch training used in training FFNNs. It can be used with updating weights on each word as well as with updating after a block of words. The multiple streams are typically given from a single training file, which is read at multiple points, e.g. with four streams, the training corpus consists of four streams, starting at words number 0, $N/4$, $N/2$ and $3N/4$ (0-based). With multistream learning, the input words (their 1-of-K encodings) may be stacked as column vectors into a single matrix $\mathbf{X}$. Then, assuming the softmax function operates column-wise, it is possible to express the forward steps of

a simple recurrent network as follows:

$$\mathbf{H}_t = \sigma(\mathbf{U}\mathbf{X} + \mathbf{W}\mathbf{H}_{t-1}) \tag{3.14}$$

$$\mathbf{Y}_t = \text{softmax}(\mathbf{V}\mathbf{H}_t) \tag{3.15}$$

The effect of this altered operation is two-fold. At first, the matrix multiplication can be computed in an optimized way, reducing the total time required for processing the whole training corpus. Secondly, when the network operates on several streams in parallel, it can not adapt to a specific topic and is forced to learn general dependencies in the data.

3.4 Implementation

To have control over the experiments to follow, a language modeling toolkit was implemented. Python was used as the programing language and the computational model was defined using the Theano toolkit [1] [9].

Theano is a toolkit for specification, compilation and execution of symbolic computation graphs. Therefore is is straightforward to express the computation of the model.

On the other hand, Theano makes it impossible to forward pass over every word just once and then back-propagate the errors from target word for several steps into history. Therefore, a mini-batch setup is accustomed: A sequence of several words is presented to the model at once. The forward pass is computed over all these word and errors are then backpropagated simultaneously from all of the words. The hidden state is passed around from one sequence to another.

This way, errors from different words are back-propagated for different number of steps. In order to ensure a certain minimal depth of backpropagation, a few words are added at beginning of each sequence. These words come from the previous sequence, thus the sequences do overlap. However, the objective function is computed only as the average of errors on the words from current sequence, avoiding duplication of targets.

The tool has a command line interface, exporting all hyperparameters. This allows running experiments from outer environments, e.g. shell scripts. A framework for running parallel experiments on an SGE cluster was built, utilizing this property.

Three classes were separated from the source code: A simple wrapper for vocabulary, handling unknown words in a defined manner. Class representing SRN model. This class not only defines the computation done by the model, but also serves a base class. Therefore, it implements utility abilities such as wrapping Theano objects into user-callable methods or storing the model to hard drive. Finally, a class representing SCRN inherits from the SRN-representing class, defining only the differences from the basic model.

To avoid segmenting always into same sequences, half-length sequences at the beginning of epoch is presented on every other epoch. E.g. the first epoch begins with sequences 1–20, 21–40, ..., the second begins with sequences 1–10, 11–30, 31–50, ... For third epoch, sequencing from the first is used and so on.

Like the `rnnlm`, gradient clipping is implemented, with clipping threshold fixed to 15. As explained in Section 3.2, this is sufficient measure for suppressing the exploding gradient problem.

Hierarchical softmax is used in no form, because the main issue with experimental results was their accuracy, not the speed.

Chapter 4

Performance Overview of the Studied Baseline Models

In the following chapter, the principal techniques from previous chapters are assessed in practice. Most of the chapter is dedicated to the investigation of the SRN model. A number of experiments was performed, exploring the behaviour of the networks under different conditions. The usual scheme for publishing experimental results consists of selecting the range of the hyperparemeter to explore, plotting the results and finally selecting the best value. Sometimes, only the best value is picked, without even stating the parameters used [3]. In this chapter, complete results for each experiment are stated, including other characteristics where appropriate. Therefore, this chapter can serve as a comprehensive test overview for Theano-based implementations of SRN.

No experiments were done with the LSTM model, as the proposed technique (see Chapter 5) does not build on top of it and the model itself is rather complicated, thus it is a big step away from the core of the work.

Furthermore, implementation details are specified in order to capture whole process of implementing SRN in detail and to allow for future replication of the stated results.

4.1 Penn Treebank Dataset

Penn Treebank (PTB) is a subset of the Wall Street Journal corpus[1]. It has been hand-annotated for grammar categories at University of Pennsylvania, thus its name. Grammar annotation allows for application of models that are aware of linguistic properties of the language.

There is a widely used preprocessed version[2]. The vocabulary is reduced to the 9999 most common words in this version and the rest of the words in the is replaced by an `<unk>` token. It is generally accepted, that this token for rare words is considered to be a regular word, i.e. predicting the `<unk>` correctly improves the performance as much as predicting any other word.

In this preprocessed version, sentence boundaries are captured only implicitly as line-breaks. It is necessary to make these explicit for a successful training of a LM. In this work, it is done by adding a `</s>` token to the end of every line. Other implementations do this as well [36] [37], however it is done at the level of reading the training file and it is

[1] LDC item number LDC99T42, https://catalog.ldc.upenn.edu/LDC99T42

[2] http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz

not a documented feature. Preliminary experiments have shown, that ignoring the end of sentences hurts the network by roughly 1 bit of per word entropy.

The widely used version of PTB is divided into three parts: Training set consists of approx. 890 000 tokens in approx. 42 000 sentences. Validation subset consists of approx. 70 000 tokens in 3370 sentences. Similarly, the test set contains approx. 79 000 tokens in approx. 3700 sentences. The validation set is used in the experiments for setting the learning rate during the training, while test set is used purely for estimating the performance.

It is persistent in all the experiments, that models perform better on the test set than on the validation set. This is simply caused by the test set being more similar to the train set. Nevertheless, the results on both these sets are very consistent.

4.2 Simple Recurrent Neural Networks

A thorough examination of the SRN performance on the PTB dataset is captured in this section. The experiments are organized from exploration of fundamental parameters, e.g. the number of hidden units, towards the less influential ones such as the weight of the regularization.

Results are reported on training (red curves), validation (blue curves) and test set (purple curves). Furthemore, the error on training data was collected during the last epoch of training. This is error is referred to as *training-dynamic* (green curves). It is always better than the error obtained on the training set in a static manner, because the network is adapting to the text on-the-fly.

4.2.1 Number of hidden neurons

An initial experiment is focused on the number of hidden neurons. The purpose of this experiment is twofold: At first, it gives a general overview of the model performance. Secondly, this experiment examines the trade-off between learning speed and final error.

The following parameters were picked for this experiment: Weights were updated once every 4 words, training was done on 4 streams in parallel. The gradients were back-propagated 10 words into the history. The initial learning rate was set to 0.1.

The results of the experiment are captured in Figure 4.1. The observed errors are smooth in the number of hidden units, which is an expected result. We can see that starting from certain number of hidden units, there is only little improvement reached by adding more. Therefore, the following experiments have not been carried out with more than 200 hidden units.

The slight degradation of performance on the validation and test datasets at 300 hidden units is not consistent with published results. It is possible, that it is purely due to some noise introduced by the random initialization. The other possible explanation is, that it corresponds to a local maximum of the overtraining – the training error reaches optimum here.

4.2.2 Depth of backpropagation and updates frequency

In the next experiment, the combined effect of backpropagation depth and frequency of weight updates is investigated. For brevity, the frequency of updates is referred to using the `bptt-block` parameter of the learning. The value of the `bptt-block` expresses after

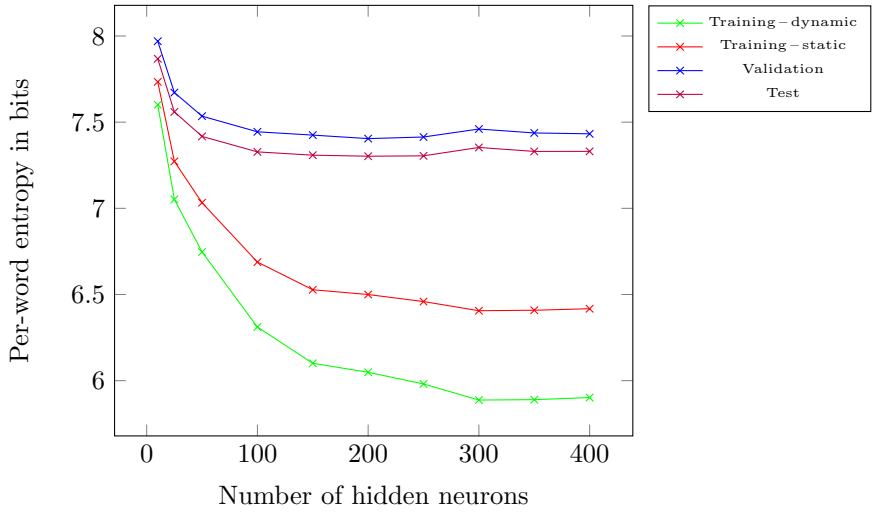
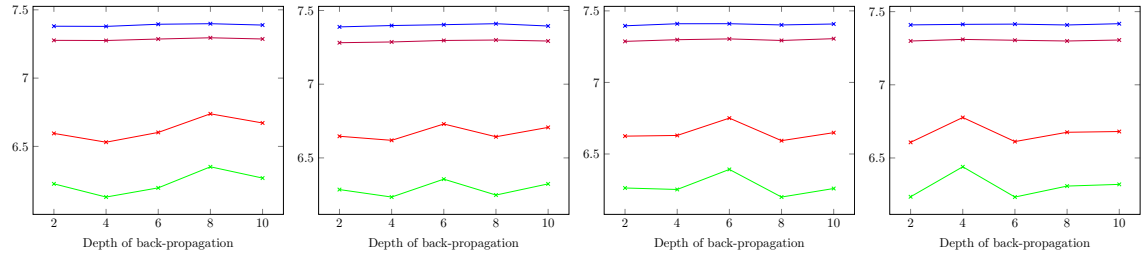


Figure 4.1: Performance of simple recurrent network (SRN) with respect to the number of hidden units. Results are average of 3 networks with different random initialization. Mean absolute difference from the median result is 0.02 bits.



(a) `bptt-block=4` (b) `bptt-block=6` (c) `bptt-block=8` (d) `bptt-block=10`

Figure 4.2: Comparison of simple recurrent network (SRN) performance with respect to how often a weights update is performed. Green is the error obtained while training; red is the error on the training dataset obtained with fixed weights; blue and purple are validation and test errors respectively.

how many words the update is performed. For this experiment, the networks have 200 hidden units. The initial learning rate is set to 0.1.

The results of the experiment are captured in Figure 4.2. For the 4 investigated values of `bptt-block`, the error is observed as dependent on the depth of backpropagation though time.

The errors on the validation and testing dataset do not exhibit any significant changes. It can be explained by examining details of the propagation: The average depth of the back-propagation of errors from a given target word is equal to the average of back-propagation depth and `bptt-block`. Therefore, the worst average depth of back propagation is 5 (updates once per 4 words, additional backpropagation 2 words). As Mikolov has shown [36], the SRN does not benefit much from back-propagating more than 6 words into the history. Therefore we can see that except the worst case, gradients are back-propagated for sufficient number of steps.

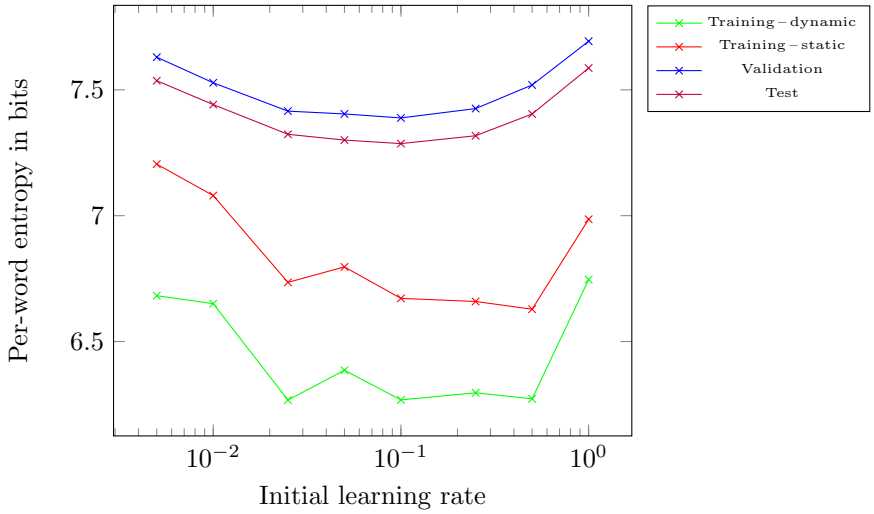


Figure 4.3: Performance of a simple recurrent network (SRN) with respect to the initial learning rate. Results are average of 3 networks with different random initialization. The rough results for training set can be simply explained by effects of the random initialization and noise in the training data.

4.2.3 Initial Learning Rate

In the following experiment, the initial learning rate is investigated. Networks in this experiment have 200 hidden units. Weights are updated every 4 words and error gradients are back-propagated for 10 words.

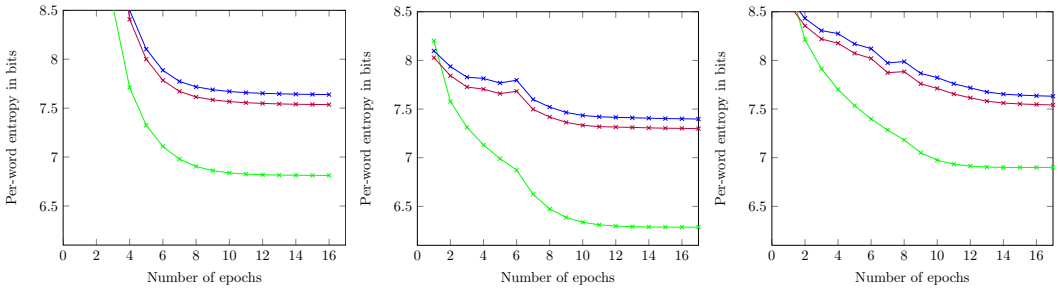
The results of the experiment are captured in Figure 4.3. Considering the validation and test set, the parameter is well-behaved. The graph is very smooth in the semi-log domain and we can see a single local minimum around 0.1. This result is in line with values published in experimental papers.

The performance on the training set is less smooth, which is not surprisive – as the model tries to optimize its performance directly on the training corpus, its performance on it is likely to sensitive to certain hyperparameters. However, the general trend also suggests that best values lies somewhere between 0.02 and 0.5.

Furthemore, the effect of different initial learning rates is examined at the level of error improvements during training of a single network. Three representative examples are shown in Figure 4.4. The training of all the models can be divided into two stages: In the first stage, the initial learning rate is kept and the network is learning its parameters. Once the validation error gets worse than in the previous epoch, the learning switches into the second stage. During this stage, learning rate is divided after every epoch, thus the errors converge to a local minimum.

With the initial learning rate 0.1 (Fig. 4.4b), the learning progress is split roughly into halves. Also, we can see the model converging quite fast, which was consistent with the other runs. There is a simple explanation for the train error being worse than the validation and test errors after the first epoch: As the train error is computed on-the-fly, it is biased by the start of the training, when the model is giving random predictions and the error is very high.

When the initial learning rate was set to a too high value (Fig. 4.4a), the model did not converge in the first stage. Therefore it effectively learns only during the second stage.



(a) Initial learning rate = 1. (b) Initial learning rate = 0.1. (c) Initial learning rate = 0.01.

Figure 4.4: Typical learning progress for different initial learning rates. Reported are results on test set (purple), validation set (blue) and training set (green). Note, that the results on the training set are computed during the training, i.e. while the weights were being adapted. The missing results for the initial learning rate 1 were just too bad – the model did hardly converge. The observed results were reached when the learning rate was already halved between epochs.

This seems to prevent the model from reaching a high-quality local optimum. One possible reason is, that the network did not have an opportunity to move by a large vector in a reasonable direction in the parameter space during the first stage.

With the initial learning rate smaller than optimal (Fig. 4.4c), we can see that the convergence is slower. The model reaches better error during the first stage, which is in line with theoretical expectations. When the learning rate is smaller, the model can dive into narrower ravines of the error function. On the other hand, the model eventually reaches an optimum that is a little bit worse. The probable explanation is the same as with initial learning rate too big – the model did not move far enough during the first stage. In this case, it was so because of the model was concentrating on fine details of the error landscape.

4.2.4 Effects of the L2-regularization

Next, the importance of the L2-regularization is assessed. As defined in Eq. (2.13), the weight of the regularization is ruled by a single parameter β. The original publication on the topic [36] claims, that applying regularization in training SRN is mainly good for numerical stability reasons and it is not important when the computation is done double-precision. This issue is not discussed in follow-up publications.

For this experiment, networks with 200 hidden units were trained. Weights were updated once every 4 words and gradients were backpropagated for 10 timesteps. The initial learning rate was set to the found optimal value of 0.1.

The results are captured in Figure 4.5. It is natural to follow the graph from its left end: With the β parameter close to zero, the model is optimizing only the error on the train data. Therefore, we can take the left-most results as baseline. In the region till 10^{-6}, the regularization has hardly any effect.

Then the training error goes briefly down only to take off later. The improvement on the training data is can be explained by suppressing the adaptation effects:

The training corpus consists of several sections, where certain topics are dominant. Thus the network adapts to the given topic during the training, as can be seen from the error difference between adaptive (green curve) and nonadaptive (red curve) processing of

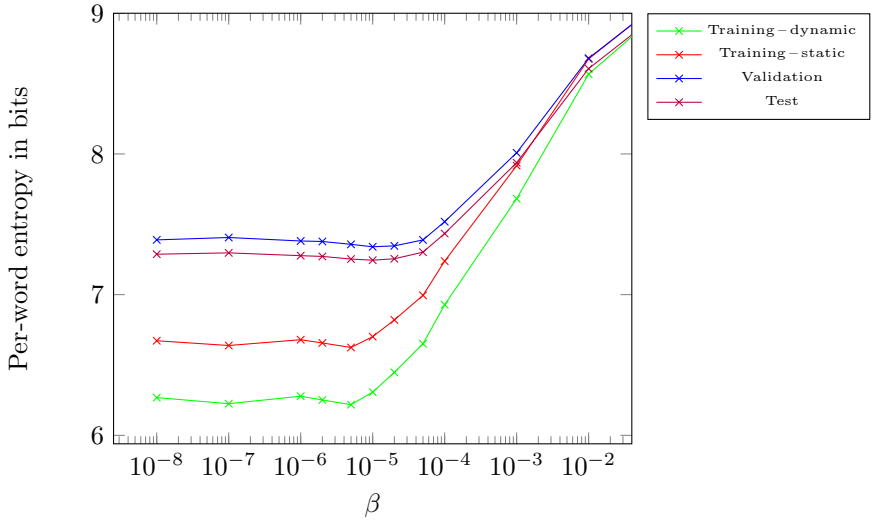


Figure 4.5: Performance of a simple recurrent network (SRN) with respect to the weight of the L2 regularization penalty. Results are average of 3 networks with different random initialization. The interesting part is the improvement on training set around $5 \cdot 10^{-6}$ and the improvement on the test and validation set around $2 \cdot 10^{-5}$.

the training corpus. But when the regularization weight is increased, the network can not over-concentrate on the current topic. Therefore, better performance is achieved on the sentences, which do adhere to the current topic very well.

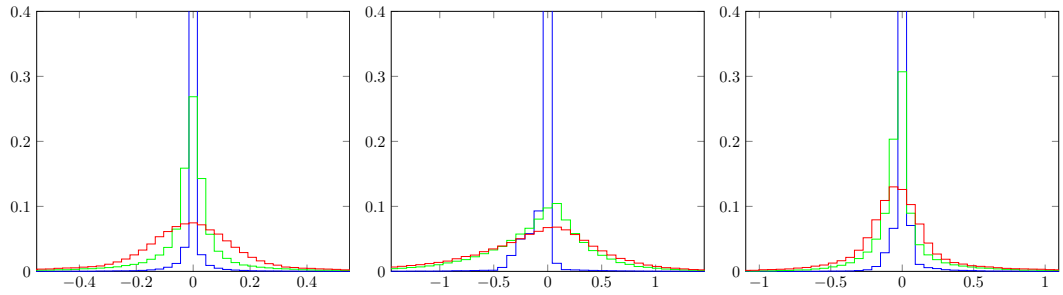
The usual effect of regularization is observed in the region from $2 \cdot 10^{-6}$ to $5 \cdot 10^{-5}$: The gap between the training and test error is closing and it is caused not only by the training error increasing, but also the validation and test error decrease bellow the baseline. This signals clearly that the networks are subject to overfitting, although its degree is small.

Once the weight of the L2 regularization passes 10^{-4}, the overall effect is very negative, as errors on all the datasets increase rapidly. Nevertheless, this is well in line with theoretical expectations – as the weight of the regularization term increases, the optimized function becomes very different from the error landscape itself.

It is straightforward to observe the effect of L2-regularization directly at the level of model parameters. Figure 4.6 shows histograms of weights under different values of β. We can see that under very loose regularization (red curves), the weights are nearly normally distributed. As the regularization weight increases, the weights in the networks are distributed more tightly around the zero.

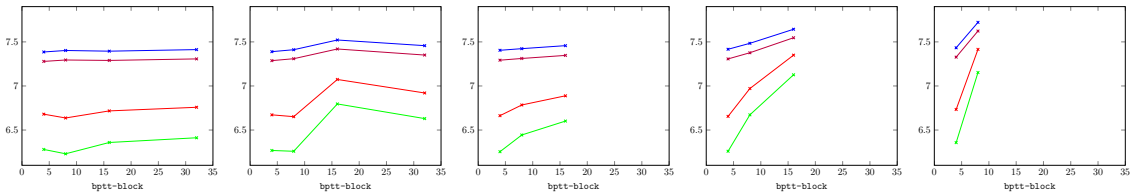
An interesting issue arises with the recurrent weights: Regardless of regularization, these weights are skewed towards the negative values. As the regularization becomes tighter, the network drops most of the positive weights, but retains a comparatively vast number of negative ones.

There are several possible reasons behind: At first, the negativity of weights implies oscillation. This is a plausible reason, since oscillation in several hundred dimensional space means jumping around. And it is reasonable for the network to expect the next word to be positioned in a different area of the history representation space, e.g. an article – a noun – a verb etc. The second reason is, that the history representation is used rather to suppress some aspects of input words than to amplify others. However, there is no explanation on why it should do so.



(a) Input mapping **U**. (b) Recurrent weights **W**. (c) Output mapping **V**.

Figure 4.6: Distribution of weights in the simple recurrent network (SRN), as dependent on the weight of L2 regularization. Distributions are plotted for β equal to: 10^{-8} (red), $2 \cdot 10^{-5}$ (green) and 10^{-3} (blue). Refer to Figure 4.5 for the impact on the performance. Histograms are computed from three randomly initialized networks. For the recurrent matrix, the histogram is computed from all weights. Subsampling down to 120 000 weights was used in the case of input and output mapping. In all cases, the zero-most bin with tight regularization goes up to around 0.8.



(a) 2 streams (b) 4 streams (c) 6 streams (d) 8 streams (e) 10 streams

Figure 4.7: Exploration of SRN behaviour with respect to number of learning streams and frequency of updates. The experiment was also conducted for 12 streams, but the result were very similar to the 10 streams, just a little bit worse. Weight updates after 4, 8, 16 and 32 words were tried for all explored numbers of streams. However, as the number of streams grew, not all models have successfully converged, thus the missing values.

A slight skew towards negative values is apparent also in the output mapping. However, it diminishes as the regularization gets tighter.

4.2.5 Multistream Learning

In his thesis, Mikolov [36] claims, that it is very beneficial to perform updates of weights often. Therefore, another experiment was focused at multistream learning. I have examined number of streams in combination with frequency of updates, as these are the two parameters governing number of updates per epoch.

In this experiment, networks were 200 hidden units wide. Gradients were backpropagated for 10 words into history.

Results of the experiment are summarized in Figure 4.7. We can see that once the product of the number of the streams and the `bptt-block` exceeds 20, the performance degrades dramatically. With lower number of streams and frequent updates, the obtained error seems quite robust to the particular setting. The error increase with 4 streams and `bptt-block`=16 seems quite random, it is possible to be introduced by some special char-

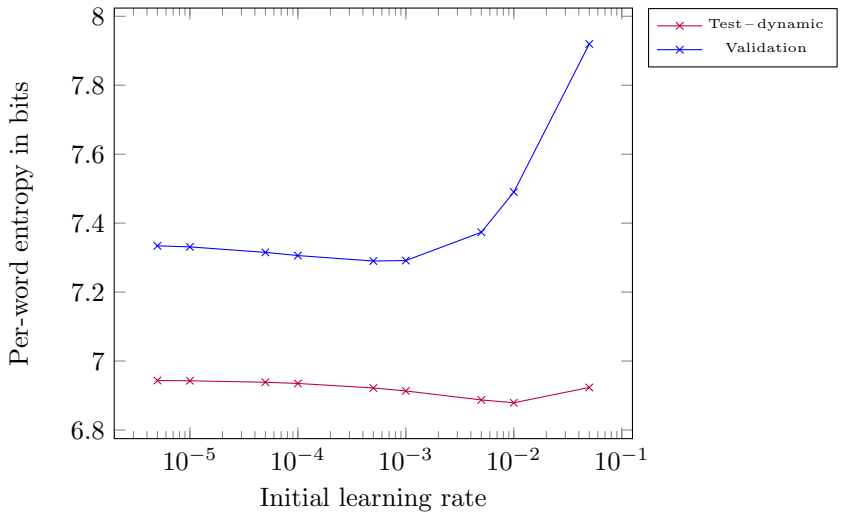


Figure 4.8: Dynamic performance on the test set. For reference, the final performance on the validation set is reported as well. These results are computed on a single input model.

acteristics of the data.

4.2.6 Dynamic Application of Networks

Finally, I have evaluated the network dynamically. For this experiment a network from the experiment examining weight of L2-regularization was taken. It was the one with best performance on the validation set. Then a single epoch of training on the test set is run. I have tried a range of different learning rates, to find optimal setup. The original training was ended with learning rate around $5 \cdot 10^{-6}$.

Result of the experiment is captured in Figure 4.8. It is obvious that best results are obtained with learning rate around 0.01. With lower learning rate, the network can not adapt enough. With higher learning rate, the network may leave the valley of error function reached during the original training.

The performance on the validation set is also interesting: Overall, the fine tuning of the model to the test set hurts the performance. With learning rate around 10^{-3}, this decrease in performance is smaller, because the network was just a little pushed from its original local optimum, which has similar effect to applying regularization. When the learning rate gets too big, the performance on the validation set is completely degraded, simply because the network adapts too much to a different error landscape.

4.2.7 Simple Recurrent Network Summary

To sum up the the experiments with the SRN models, I have compared the achieved results with the Mikolov's original implementation `rnnlm`. The most interesting results are summarized in Table 4.1.

We can see that this Theano-based implementation has not reached the best published results. As all available hyperparameters have been extensively explored, the only difference remaining is the way of passing through the trainined set. Since the L2 regularization has helped a lot (removes 30 % of the difference) and has been claimed by Mikolov not to bring any gains, we can assume that his originial implementation serves as strong regularizer.

Table 4.1: Performance of different models on the test set. Upper and lower bound is given by an external implementation of RNN LM and a smoothed 3-gram model respectively.

Model	Entropy	Perplexity
Good Turing 3-gram	7.38	165.2
Basic experiment, 100 neurons	7.32	159.8
Basic experiment, 200 neurons	7.30	157.6
$\beta = 10^{-5}$, 200 neurons	7.24	151.2
$\beta = 10^{-5}$, 200 neurons, dynamic	**6.87**	**117.0**
`rnnlm` toolkit, 200 neurons	**7.10**	**138.4**
`rnnlm` toolkit, unknown number of neurons	6.94	123.2

4.3 Structurally Constrained Recurrent Network

The paper introducing SCRN comes also with a link to public implementation[3]. However, it is implemented in Torch, which is a library for tensor computation in Lua scripting language. Since Theano was selected as the tool for implementation in this work, the SCRN model has been reimplemented[4].

The paper also comes with reasonably detailed description of hyperparameters used for training the model[37]. Therefore, the first natural experiment was focused on replicating the published result. However, the models did not converge at all in my case.

Taking experience from experiments with SRN, the most likely reason of the divergence is the combination of a rather big number of streams (32) with updates after 5 words and a very long backpropagation trough time (50 steps).

Therefore, I have decided to try an experiment with modest setting of these parameters. The following experiment has been done with 4 streams, updating after every 4th word. The networks were 200 hidden neurons wide. Backpropagation was done for 10 steps only. This can be expected to decrease the effect of introducing slow neurons, however an improvement should be observed anyway, since we expect regular neurons to learn from histories up to 6 words only.

To suppress effects of random initialization, this experiment was run as a search over the number of slow neurons. Results of the experiment are summarized in Figure 4.9. No substantial improvement was observed, and definitely not a consistent one. Overall, the model seems to overtrain more than a SRN and the results are worse than the published ones. Therefore, the implementation technique where a single hidden state is computed only once during the epoch really serves as a strong regularizer.

I did not perform experiments with explicit regularization, as the author did not mention it at all and the reached results were worse than a comparable SRN anyway.

[3] https://github.com/facebook/SCRNNs
[4] See Appendix B for code.

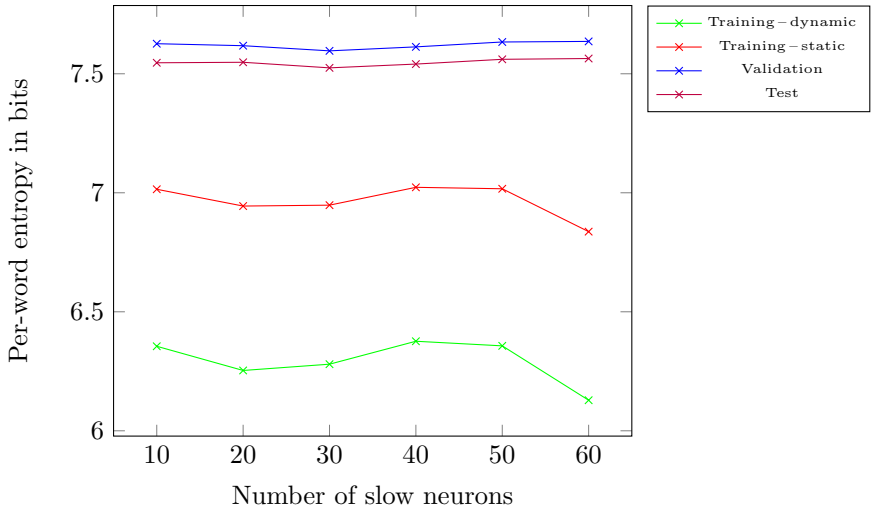


Figure 4.9: Performance of a Structurally Constrained Recurrent Network (SCRN) with respect to the number of slow neurons. Results are average of 3 networks with different random initialization. We can see that the slower neurons have hardly any effect on the test and validation performance. Slight improvement on the train set can be observed, but not in smooth and interpretable way.

Chapter 5

Forcing Sparsity in the Recurrent Weights Matrix

As explained in Subsection 2.5, the SCRN model can be interpreted as a simple recurrent network with constrained values in the recurrent weights matrix. These constraints are supposed to help the network to focus on different aspects of the text with different parts of the parameters.

Sparsity in recurrent networks is proposed by Bengio et al. as well [3]. In their approach, the sparsity is not an inherent property of the model, but is forced by L1 regularization. However, they use ReLU as the nonlinearity in the hidden layer.

Another example of proposing sparsity can be found in a more theoretically based work of Sutskever et al. [46]. They advocate initializing the recurrent weights in the form of a sparse matrix. Their reasoning is that a network initialized this way would not change its hidden state so rapidly, so it is easier to back-propagate the error and distribute the adjustments to the responsible parameters.

Finally, there is a quite successful model called Echo State Network (ESN) used in general sequence prediction [26]. This model is, in its topology, very similar to the SRN. However, only the output weights are learned, thus the training is computationally very simple, as no backpropagation takes place. The rest of the weights are simply set during creation of the network. Moreover, the recurrent weights are initialized as being very sparse, e.g. 15 input connections per neuron for a hidden layer with a few hundreds of neurons.

All of these examples suggest, that it is beneficial to keep some of the recurrent connection at zero. Thus, a simple extension to this idea is studied in this chapter.

5.1 Randomly Sparsed Recurrent Neural Network Model

The proposed model is based directly on the SRN model. The change is introduced to the computation of the hidden state. The original equation (2.14) is replaced by its enhanced version:

$$\mathbf{h}_t = \sigma(\mathbf{U}\mathbf{x}_t + ((\mathbf{W}_m \odot \mathbf{W})\mathbf{h}_{t-1}) \tag{5.1}$$

The introduced matrix $\mathbf{W}_m$ is a binary mask and the $\odot$ operation represents entry-wise multiplication. Thus, the only ability of $\mathbf{W}_m$ is to turn single entries of $\mathbf{W}$ on or off. This matrix is set as a fixed parameter of the model and can thus be initialized by specific shape or randomly.

The introduction of $\mathbf{W}_m$ allows to turn the network sparse. In this work, random initialization of $\mathbf{W}_m$ is studied, with a focus on the behaviour of the networks when there are only a few entries set to one. Therefore, the proposed model is called Randomly Sparse Recurrent Neural Network (RS-RNN).

The governing parameter is then the amount of nonzero elements in $\mathbf{W}_m$. Its normalized value is called *density*. Note, that for density equal to 1, the RS-RNN is equivalent to the SRN.

This approach is similar to the dropout technique described in Section 2.2.3. It represents a dropout applied only to the recurrent connections. The difference from the standard form of dropout consists of two concepts: At first, the RS-RNN is simpler because the mask of dropping out is fixed for all the training. Secondly, dropout turns off whole neurons, while in RS-RNN, only specific neuron-neuron connections are set on or off.

However, dropout is rarely used for recurrent connections. The main reason is that it is usually discussed in context of the LSTM model. In the LSTM model, the network learns to protect its memories and therefore grows dependent on their precise values. So it is very difficult for the network to learn what to expect from any specific memory in the presence of dropout, where only ~20–30 % of the neurons are kept from previous timesteps.

Thus in practise, dropout is used only for the regular non-recurrent connections [1]. This approach is also supported by some publications [47] [44]. There were some experiments suggesting, that with a careful setting, even a dropout on the recurrent connections may work, although not as well as dropout at other places of the architecture [11].

In his recent paper, Gal has investigated the application of dropout in RNNs interpreted as Bayesian networks [17]. Supporting my approach, he shows that using the same mask for all timesteps is correct way to approach dropout in recurrent networks.

5.2 Performance of the Randomly Sparse RNN Model

As a quick assessment of the potential of the RS-RNN model, a simple experiment with 100 hidden units was performed. The random mask was randomly initialized so that 20 %, 40 %, 60 %, 80 % and 100 % of its entries were set to one. The networks were trained with initial learning rate 0.1, updating the weights every 30 words and backpropagating the gradients for 5 words into the history.

The results of this experiment are captured in Figure 5.1. The error on the validation and test set is slightly increasing as the recurrent weights grow sparse. Taking the analogy with the experiment exploring the effect of the number of hidden units, this is a direct impact of decreasing the number of learnable parameters of the model. However, the decreasing error on the training set does not follow this explanation. This improvement can be explained using similar reasoning as Sutskever et al. did for their sparse initialization [46]: Having the recurrent connections sparse makes it easier for the network to learn temporal dependencies in the data, because the hidden state does not change so rapidly with every timestep.

Thus, the decreased performance on the validation and test set may be caused by the model fitting the training corpus better.

[1] Personal communication with R&D staff from Seznam.cz.

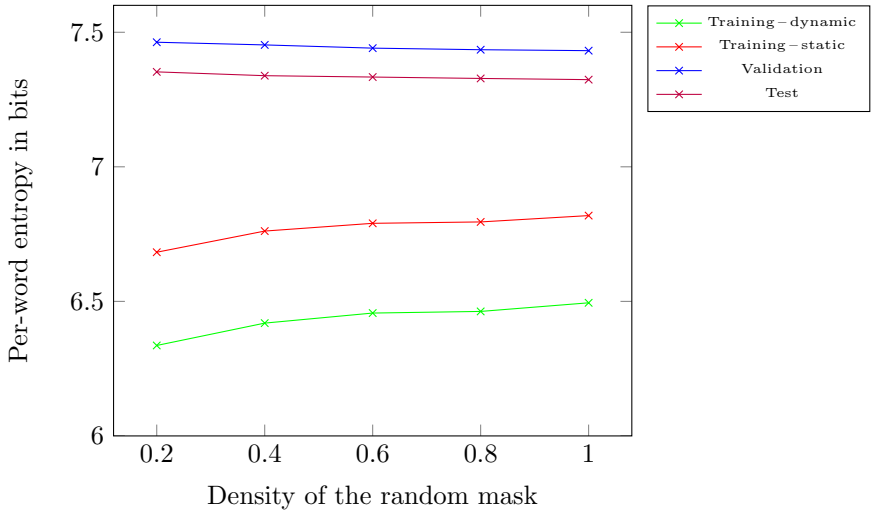


Figure 5.1: Performance of a Randomly Sparse Recurrent Neural Network (RS-RNN) with 100 hidden neurons, with respect to the density of the random mask. Thus sparser models are in the left. Results are average of 9 networks with different random initialization.

5.2.1 Interpolation of Several Randomly Initialized Models

Since the RS-RNN seems to learn the training corpus better than a dense SRN, the idea of combining such models becomes tempting. For this purpose, wider models with 200 hidden neurons were trained, with the rest of the learning parameters kept same as in the previous experiment. The individual performance of these models is captured in Figure 5.2.

The combination of models is done as an unweighted linear interpolation (5.2), also called *posterior combination*. The motivation behind this is that the models should be in principle equal, so training interpolation weights would just exploit differences introduced by noise.

$$p(w_t|p_1^{t-1}) = \frac{1}{N}\sum_{i=1}^{N} p_i(w_t|p_1^{t-1}) \tag{5.2}$$

The results of the interpolation are captured in Figure 5.3. We can see that when the mask has as few as 20 % zero entries (density = 0.8), the result of combination is much better than combination of SRN models (density = 1). The improvement over a baseline – the average of single systems with test entropy 7.28 – has been quantified in Table 5.1. We can see that the relative improvement of the gain of the RS-RNN compared to the SRN is approximately 30 %.

5.2.2 Applying the L2 Regularization on Randomly Sparse RNN

The other way to exploit the better performance of RS-RNN on the training data is to resctrict it with regularization. To explore this possibility, an experiment has been conducted with RS-RNNs. Taking experience from experiments with regularization on SRN (see Section 4.2.4) only a narrower range of regularization weights was explored.

The effect of the regularization is pictured in Figure 5.4. We can see that the general behaviour of RS-RNN under L2 regularization is similar to the SRN. The maximum gain, obtained at $\beta = 2 \cdot 10^5$, is similar to the gain of the SRN. However, since the models

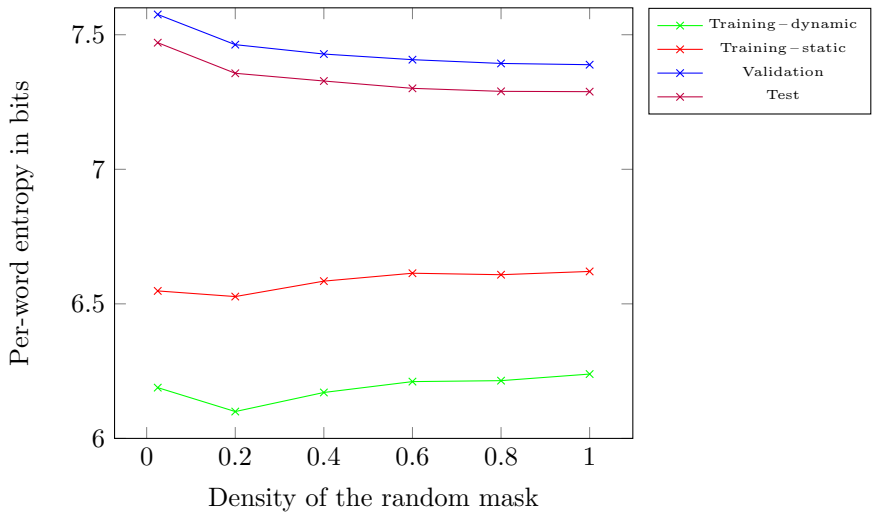


Figure 5.2: Performance of a Randomly Sparse Recurrent Neural Network (RS-RNN) with 200 hidden neurons, with respect to the density of the random mask. Results are average of 9 networks with different random initialization. We can see that the general trend is very similar to the networks with 100 hidden neuron (see Figure 5.1). When the density drops too low (5 %, leftmost entry), we can see that overall performance gets worse. The main reason is a severe decrease of the number of parameters in the recurrent layers.

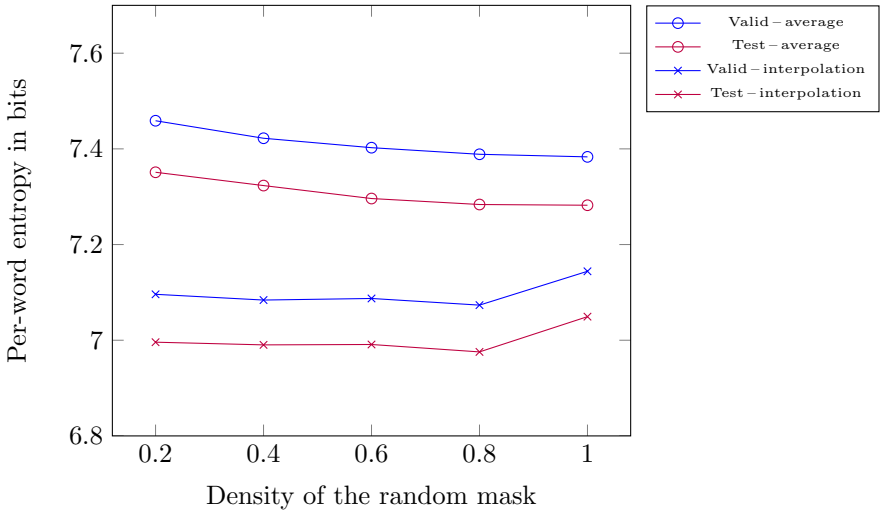


Figure 5.3: Effect of interpolation of randomly initialized models as dependent on the density of random mask, compared to the performance of individual models. All models have 200 hidden neurons and for every density, nine models have been trained and combined.

Table 5.1: Performance of models interpolation on the test set as dependent on the density of the random mask. The improvement over baseline is reported on entropy.

Density	Entropy	Perplexity	Improvement over baseline
1.0	7.05	132.5	3.2 %
0.8	6.97	125.4	4.2 %
0.6	6.99	127.1	4.0 %
0.4	6.99	127.1	4.0 %
0.2	7.00	128.0	3.8 %

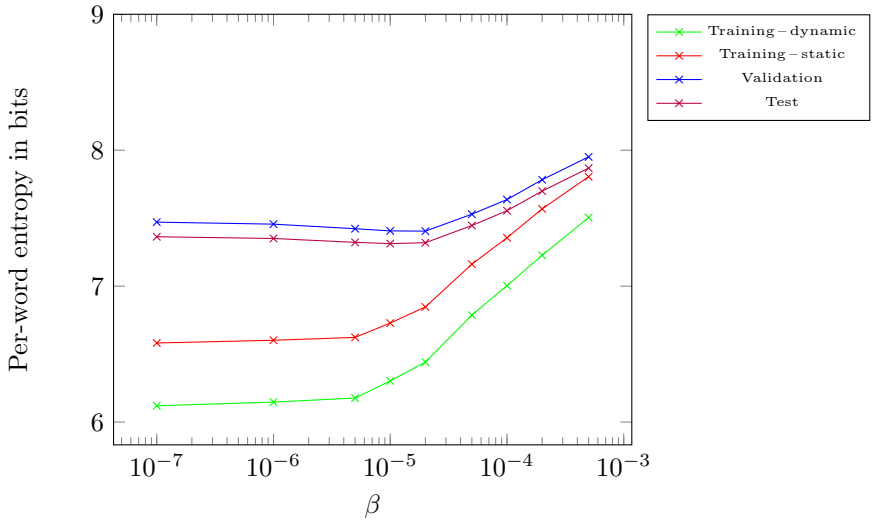


Figure 5.4: Effect of the L2-regularization on models with density=0.2. All models have 200 hidden neurons and for every weight of regularization, nine models have been trained and combined.

perform much worse without regularization, the final result is comparable to a SRN without regularization.

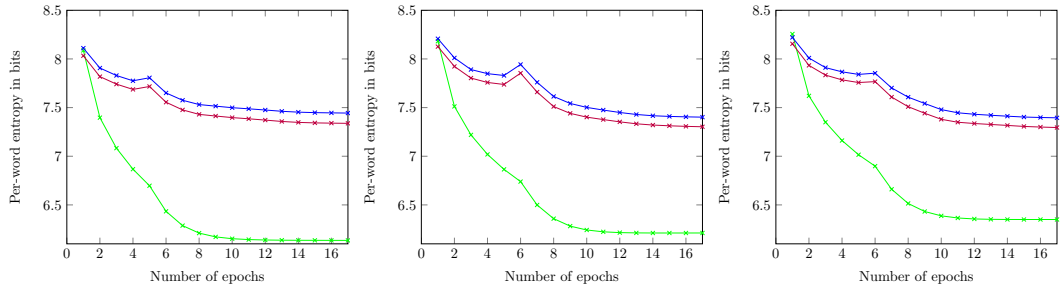
The intersting observation is, that even with quite tight regularization, the gap between dynamic and static performance on the training set remains significant.

5.3 Additional Properties of the Randomly Sparse RNN Model

Since the restricting mask $\mathbf{W}_m$ rather significantly alters the performance of the model, it is worth investigating, how the inner workings of the RS-RNN differ from a similar SRN.

As the first case, the learning progress of a single model is examined. For this, three models of different density were taken from the experiment with 200 hidden neurons. These models are not special in any respect.

The respective learning progress is plotted in Figure 5.5. We can see that the selected models improve very similarly. The only significant difference is the performance degradation of the model with density = 0.6 in the sixth epoch, but that may well be caused by some random effect. Also, it has no further effect because the training procedure performs a rollback to the previous model when the validation error increases.



(a) Density = 0.2. (b) Density = 0.6. (c) Density = 1.

Figure 5.5: Typical learning progress for different density of the random mask. Reported are results on test set (purple), validation set (blue) and training set (green). Note, that the result on the training set are computed during the training, i.e. while the weights were being adapted.

Furthermore, the distribution of weights is examined. For this analysis, weights from all models with a given density were accumulated. I believe that by doing so, no important quality of the data is lost. Furthermore, only those of recurrent weights are taken into account, which are active.

Histograms of weights in all the matrices are presented in Figure 5.6. We can see that the input mapping and output mapping are distributed nearly normally. The recurrent weights are much more have a higher variance.

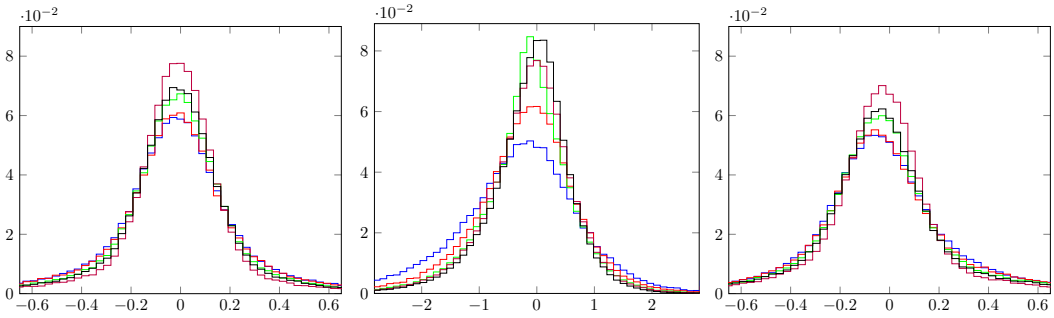
The same pattern can be observed for the weights in both $\mathbf{U}$ and $\mathbf{V}$: The sparser the mask, the wider the distribution. The exception is the fully connected network (black curve), for which the weights are distributed a little wider.

The recurrent weights follow a similar pattern, but loosely. The distribution of weights for middle density (0.6, green curve) is nearly identical to distribution of weights in a dense network (black curve).

The recurrent weights also are biased towards the negative values, like in the case of SRN (see Subsection 4.2.4). However, the bias towards negative values gets more significant as the weights get sparser. We can assume that it is important for the network to retain a significant portion of its negative weights.

With some of the sparse models a question may arise, how many hidden neurons from the past do actually affect a particular one? We can see, that due to transitivity, all of them contribute: Let us assume an RS-RNN with the density of the mask equal to 0.2. Then, a hidden neuron is – on average – connected to 40 hidden neurons in the previous step. However, looking two steps back, this particular neuron should be connected to $40 \cdot 40 = 1600 \gg 200$ neurons. Thus we can safely assume, that it is effectively connected to all of them. This has been empirically verified on all generated masks.

We can expect this to hold for masks down to density $d = 1/\sqrt{N}$. Therefore, the gains achieved by making the matrix sparse clearly do not come from lower level of connectivity between hidden neurons, but slower change of state. Thus I suggest that the improvements reached are obtained by the model being easier to learn, due to easier distribution of gradients in the first step of backpropagation through time.



(a) Input mapping **U**. (b) Recurrent weights **W**. Note the different axes scale. (c) Output mapping **V**.

Figure 5.6: Distribution of weights in the Randomly Sparse Recurrent Neural Network (RS-RNN), as dependent on the density of the random mask. Distributions are plotted for density equal to: 0.2 (blue), 0.4 (red), 0.6 (green), 0.8 (purple) and 1 (black). Refer to Figure 5.2 for impact on performance. Histograms are computed from nine randomly initialized networks. For the recurrent matrix, the histogram is computed from all nonzero weights. Subsampling down to 120 000 weights was used in the case of input and output mapping.

Chapter 6

Conclusion

In this diploma project, I have investigated learning procedures of recurrent neural networks (RNN) in language modeling. Several prevalent models are presented, as well as techniques for their training. Based on this knowledge, a language modeling tool was implemented in order to experiment with the models. Using this tool, the Simple Recurrent Network (SRN) model was intensively tested in a range of situations. Also, the Structurally Constrained Recurrent Network (SCRN) was implemented and tested. A novel enhancement of the SRN was proposed and thoroughly studied.

The downside of the Theano based SRN implementation is, that the result reached was by 2.7 % worse than the best published result. On the other hand, I have found out that L2 regularization helps to reduce this gap by 30 %. This result is contrary to previous publications on the topic, which claim that L2 regularization does not bring any improvement in performance. Finally, I have found out that the implementation eventually outperforms the best published results when evaluating the test set dynamically.

The SCRN model did not converge when using the parameters given by its authors. In a simpler setup, the result did not get better than the SRN baseline.

The conclusion from these experiments is, that the implementation details implied by Theano lead to a worse performance. Other implementations use a more direct approach, which seems less in line with the theory of gradient learning, however works as a strong regularizer.

The RS-RNN model seems to overtrain a little, compared to the SRN model. However, it has been empirically shown, that the gains on the training corpus can be effectively transformed into improved performance on a test set by a simple posterior combination. This improvement is by 30 % larger than the improvement reached by a combination of the same number of dense SRN models. This result has been presented at the Excel@FIT student conference [2].

Future work will be oriented in two directions: The RS-RNN model will be integrated into some of the toolkits that produce the state of the arts results. If the improvement will last, it should be a model improvement suitable for publication.

The other extension of this diploma project lies in combining the RS-RNN model with a Bayesian treatment of RNNs in language modeling. Since the Bayesian treatment poses a way of regularization on its own, it will be interesting to see whether the improved learning performance of the RS-RNN could be utilized in an other way than the interpolation of several models.

Bibliography

[1] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[2] Karel Beneš. Randomly Sparsed Recurrent Neural Networks for Language Modeling. In *Excel@FIT, Student Conference*, 2016.

[3] Y. Bengio, N. Boulanger-Lewandowski, and R. Pascanu. Advances in Optimizing Recurrent Networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8624–8628, May 2013.

[4] Yoshua Bengio. Learning Deep Architectures for AI. *Foundations and Trends® in Machine Learning*, 2(1):1–127, 2009.

[5] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155, March 2003.

[6] Yoshua Bengio, Pascal Lamblin, Dan Popovici, Hugo Larochelle, Université De Montréal, and Montréal Québec. Greedy Layer-Wise Training of Deep Networks. In *In NIPS*. MIT Press, 2007.

[7] Yoshua Bengio, Yann Lecun, Département D'informatique Et Recherche Opérationnelle, Universite De Montreal, L. Bottou, O. Chapelle, D. Decoste, and J. Weston (eds. Scaling learning algorithms towards ai, 2007.

[8] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning Long-Term Dependencies with Gradient Descent is Difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[9] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[10] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[11] Théodore Bluche, Christopher Kermorvant, and Jérôme Louradour. Where to apply dropout in recurrent neural networks for handwriting recognition? In *International Conference of Document Analysis and Recognition (ICDAR)*, 2015.

[12] David S. Broomhead and David Lowe. Radial Basis Functions, Multi-Variable functional interpolation and adaptive networks. *Complex Systems*, 2:321–355, March 1988.

[13] KyungHyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *CoRR*, abs/1409.1259, 2014.

[14] George Cybenko. Approximation by Superpositions of a Sigmoidal Function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, December 1989.

[15] Jeffrey L. Elman. Finding Structure in Time. *COGNITIVE SCIENCE*, 14(2):179–211, 1990.

[16] Scott E. Fahlman and Christian Lebiere. The Cascade-Correlation Learning Architecture. In *Advances in Neural Information Processing Systems 2*, pages 524–532. Morgan Kaufmann, 1990.

[17] Yarin Gal. A theoretically grounded application of dropout in recurrent neural networks. *arXiv:1512.05287*, 2015.

[18] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, pages 249–256, 2010.

[19] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep Sparse Rectifier Neural Networks. In Geoffrey J. Gordon and David B. Dunson, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS-11)*, volume 15, pages 315–323. Journal of Machine Learning Research - Workshop and Conference Proceedings, 2011.

[20] Joshua Goodman. Classes for fast maximum entropy training. *CoRR*, cs.CL/0108006, 2001.

[21] Joshua T. Goodman. A bit of progress in language modeling. *Computer Speech & Language*, 15(4):403 – 434, 2001.

[22] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A Fast Learning Algorithm for Deep Belief Nets. *Neural computation*, 18(7):1527–1554, 2006.

[23] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[24] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251 – 257, 1991.

[25] Johan Håstad and Mikael Goldmann. On the Power of Small-Depth Threshold Circuits. *Computational Complexity*, 1:610–618, 1991.

[26] Herbert Jaeger. Adaptive nonlinear system identification with echo state networks. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems 15*, pages 609–616. MIT Press, 2003.

[27] Herbert Jaeger, Mantas Lukoševičius, Dan Popovici, and Udo Siewert. Optimization and applications of echo state networks with leaky- integrator neurons. *Neural Networks*, 20(3):335 – 352, 2007. Echo State Networks and Liquid State Machines.

[28] Frederick Jelinek, Bernard Merialdo, Salim Roukos, and Martin Strauss. A dynamic language model for speech recognition. In *Proceedings of the Workshop on Speech and Natural Language*, HLT '91, pages 293–295, Stroudsburg, PA, USA, 1991. Association for Computational Linguistics.

[29] Rafal Józefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 2342–2350, 2015.

[30] Andrey N. Kolmogorov. On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition. *Dokl. Akad. Nauk SSSR*, 114:953–956, 1957.

[31] Norbert Kruger, Peter Janssen, Sinan Kalkan, Markus Lappe, Ales Leonardis, Justus Piater, Antonio J. Rodriguez-Sanchez, and Laurenz Wiskott. Deep Hierarchies in the Primate Visual Cortex: What Can We Learn for Computer Vision? *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(8):1847–1871, August 2013.

[32] Roland Kuhn and Renato De Mori. A cache-based natural language model for speech recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(6):570–583, 1990.

[33] Zachary Chase Lipton. A critical review of recurrent neural networks for sequence learning. *CoRR*, abs/1506.00019, 2015.

[34] Mehryar Mohri and Fernando Pereira and Michael Riley. Speech recognition with weighted finite-state transducers. In *Springer Handbook on Speech Processing and Speech Communication*, chapter 28. Springer-Verlag New York, Inc., 2008.

[35] Risto Miikkulainen and Michael G. Dyer. Natural language processing with modular neural networks and distributed lexicon. *Cognitive Science*, 15:343–399, 1991.

[36] Tomáš Mikolov. *Statistical Language Models Based on Neural Networks*. PhD thesis, 2012.

[37] Tomas Mikolov, Armand Joulin, Sumit Chopra, Michaël Mathieu, and Marc'Aurelio Ranzato. Learning longer memory in recurrent neural networks. *CoRR*, abs/1412.7753, 2014.

[38] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546, 2013.

[39] Tomas Mikolov, Wen tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT-2013)*. Association for Computational Linguistics, May 2013.

[40] Michael C. Mozer. Neural net architectures for temporal sequence processing. pages 243–264. Addison-Wesley, 1994.

[41] Wim De Mulder, Steven Bethard, and Marie-Francine Moens. A survey on the application of recurrent neural networks to statistical language modeling. *Computer Speech & Language*, 30(1):61 – 98, 2015.

[42] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. Understanding the Exploding Gradient Problem. *CoRR*, abs/1211.5063, 2012.

[43] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, October 2014. Association for Computational Linguistics.

[44] Vu Pham, Christopher Kermorvant, and Jérôme Louradour. Dropout improves recurrent neural networks for handwriting recognition. *CoRR*, abs/1312.4569, 2013.

[45] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[46] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In Sanjoy Dasgupta and David Mcallester, editors, *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, volume 28, pages 1139–1147. JMLR Workshop and Conference Proceedings, May 2013.

[47] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *CoRR*, abs/1409.2329, 2014.

Appendices

List of Appendices

Appendix A

Contents of the CD

```
/
├── thesis.pdf ..... Text of the thesis.
├── tex ..... LaTeX sources.
│   ├── pgf ..... Plots of the experiment results, including data.
│   └── tikz ..... Images picturing models and illustrating learning techniques.
├── pythlem ..... Python implementation.
│   ├── main.py ..... The runnable script.
│   └── model.py ..... Definition of models in Theano.
├── penn-treebank-sentences ... Penn Treebank corpus, version used for experiments.
├── sge-ptb ..... Bash framework for running experiments.
├── extractor.py Extracts results from a logs together with selected hyperparameters.
└── epochal-extractor.py ..... Collects per-epoch performance.
```

Appendix B

Definition of Models in Theano

This Appendix presents parts of the code responsible for the actual computation. For complete code, refer to Appendix A.

Note that the samples are generally stored in 3D tensors, by default indexed as `(stream, timestep, sample items)`. This implies, that when taking a 2D slice, each sample is a row vector. Thus, vectors are multiplied into matrices from the left, unlike the math definition in Chapter 2 and 3.

The class `StructurallyConstrainedRecurrentNetwork` does not have its own contructor, it uses just the one inherited from `SimpleRecurrentNetwork`.

Note that line have been broken to fit into the page.

```
def softmax3d(x):
    if x.ndim != 3:
        raise ValueError("Tried␣to␣apply␣3d␣softmax␣on␣" + str
           (len(x.shape) + "-d␣tensor"))

    e_x = T.exp(x - x.max(axis=2, keepdims=True))
    return e_x/e_x.sum(axis=2, keepdims=True)

class SimpleRecurrentNetwork:
    def __init__(self, vocab_size, hidden_config, clip_thres,
       beta, custom_mask = None):
        self._hidden_config = hidden_config
        self._init_params(vocab_size, hidden_config,
           custom_mask)
        self._init_hidden_inputs(hidden_config)

        self._seq = T.imatrix("x") # a row is a "sentence"
        self._lr = T.scalar("learning␣rate")

        _nb_targets = T.iscalar("number␣of␣targets")
        _req_h = T.iscalar("requested␣hidden␣state␣(from␣end)"
           )

        nb_seqs = self._seq.shape[0]
        seq_len = self._seq.shape[1]
```

```
        t = self._seq[:,-_nb_targets:]

        self._y = self._model_computation(self._seq,
           clip_thres) # 0 - streams, 1 - time, 2 - individual
            word probas
        y_reshaped =  self._y.dimshuffle(1, 0, 2) # 0 - time,
           1 - streams, 2 individual word probas
        hidden_output = self._hidden_output(_req_h)

        self._err = -T.sum(T.log2(
            y_reshaped[T.arange(y_reshaped.shape[0]-
               _nb_targets, y_reshaped.shape[0]).reshape
               ((-1,1)), T.arange(y_reshaped.shape[1]), t.T]
        ))

        self._l2_norm = T.sum([T.sum(x*x) for x in self.
           _params])
        cost = self._err + beta*seq_len*self._l2_norm

        self._sgd = StochasticGradientDescent(self._params,
            cost, self._lr, clip_thres)

        self._test = theano.function(
            inputs=[self._seq, _nb_targets, _req_h] + self.
               _hidden_inputs,
            outputs=[self._err, self._y] + hidden_output
        )


        self._train = theano.function(
            inputs=[self._seq, self._lr, _nb_targets, _req_h]
               + self._hidden_inputs,
            outputs=[self._err, self._y] + hidden_output,
            updates=self._sgd.get_updates()
        )

....

    def _model_computation(self, seq, clip_thres):
        self._p = self._U[seq[:, :-1]].dimshuffle(1, 0, 2)
        # dimshuffle necessary, so that time is the outermost
           axis

        def srn_step(p_t, h_tm1, rec_weights, rec_weights_mask
           ):
            x_t = T.as_tensor_variable(p_t, ndim=2) # dim 0 -
               samples from different streams, dim 1 -
               elements of their word-vectors
```

```
            h_t = T.nnet.sigmoid(x_t + T.dot(h_tm1,
               rec_weights*rec_weights_mask))
            return theano.gradient.grad_clip(h_t, -clip_thres,
                clip_thres)

        self._hs1, updates = theano.scan(
                fn = srn_step,
                sequences = self._p,
                outputs_info = [self._h0],
                non_sequences = [self._W, self._W_mask],
                truncate_gradient = -1,
                strict = True
            )

        self._hs1 = self._hs1.dimshuffle(1, 0, 2)
        self._h = T.concatenate([self._h0.dimshuffle(0, 'x',
           1), self._hs1], axis = 1)
        return softmax3d(T.dot(self._h, self._V))
......

class StructurallyConstrainedRecurrentNetwork(
   SimpleRecurrentNetwork):
    def _model_computation(self, seq, clip_thres):
        self._p = self._U[seq[:, :-1]].dimshuffle(1, 0, 2)
        self._b = self._B[seq[:, :-1]].dimshuffle(1, 0, 2)

        def step(p_t, b_t, h_tm1, s_tm1, W, P):
            x_t = T.as_tensor_variable(p_t, ndim=2) # dim 0 -
               samples from different streams, dim 1 -
               elements of their word-vectors
            l_t = T.as_tensor_variable(b_t, ndim=2) # dim 0 -
               samples from different streams, dim 1 -
               elements of their bag-of-word-vectors

            s_t = (1-self._alpha)*l_t + self._alpha*s_tm1
            h_t = T.nnet.sigmoid(x_t + T.dot(h_tm1, W) + T.dot
               (s_t, P))
            return theano.gradient.grad_clip(h_t, -clip_thres,
                clip_thres), theano.gradient.grad_clip(s_t, -
               clip_thres, clip_thres)

        [self._hs1, self._ss1], updates = theano.scan(
                fn = step,
                sequences = [self._p, self._b],
                outputs_info = [self._h0, self._s0],
                non_sequences = [self._W, self._P],
                truncate_gradient = -1,
                strict = True
```

```
    )

self._hs1 = self._hs1.dimshuffle(1, 0, 2)
self._ss1 = self._ss1.dimshuffle(1, 0, 2)

self._h = T.concatenate([self._h0.dimshuffle(0, 'x',
   1), self._hs1], axis = 1)
self._s = T.concatenate([self._s0.dimshuffle(0, 'x',
   1), self._ss1], axis = 1)
return softmax3d(T.dot(self._h, self._V_h) + T.dot(
   self._s, self._V_s))
```