Nova Universidade de Lisboa

Master Thesis

Sentiment Classification Using Tree-Based Gated Recurrent Units

Vasileios Tsakalos

supervised by

Dr. Roberto Henriques

June 7, 2017

Acknowledgements

I would like to thank my supervisor professor, Dr. Roberto Henriques, for helping me structure the report, advising me with regards to the mathematically complexity required, and for making me realise that there is no reason for much technicalities as long as I get my point across. Moreover I would like to thank my brother, Evangelos Tsakalos, for providing me with the computational power to run the experiments and advising me regarding the subject of my master thesis.

Abstract

Natural Language Processing is one of the most challenging fields of Artificial Intelligence. The past 10 years, this field has witnessed a fascinating progress due to Deep Learning. Despite that, we haven't achieved to build an architecture of models that can understand natural language as humans do. Many architectures have been proposed, each of them having its own strengths and weaknesses. In this report, we will cover the tree based architectures and in particular we will propose a different tree based architecture that is very similar to the Tree-Based LSTM, proposed by Tai(2015). In this work, we aim to make a critical comparison between the proposed architecture -**Tree-Based GRU**- with Tree-based LSTM for sentiment classification tasks, both binary and fine-grained.

Keywords : Deep Learning, Natural Language Processing, Recursive Neural Networks, Sentiment Classification

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

This paper aims to contribute to the Artificial Intelligence community by proposing a different architecture(Tree-based GRU 3.2.6) and presenting its results in comparison to Tree-Based LSTM [1](3.2.5). The conduction of this experiment is done as GRU architecture is less complicated and has less parameters to compute than LSTM. Therefore, we hypothesize that it is faster to train. GRU is a new approach, it is not determined whether it is better than LSTM or not [2], so far it is assumed that the comparison between those two models is like a comparison betweem non-linear activation functions, there is no "best" function but some non-linearities suit better some problems.

1.2 Overview of Thesis

This thesis consists of 5 chapters. The first chapter, the introduction, provides the reader with the necessary background to be able to read this report. The second chapter(2), word embeddings, is about the representation of the words on the vector space. On the second chapter we will go through the word embeddings benefits and the most efficient algorithms that are used to achieve this transformation (from words, to vectors). It is important for the reader to be aware of the fact that word embeddings are the input for the Tree-based GRU (and every other language model that we cover in this report). The third chapter(3), feedback neural networks, makes an in depth commentary of the feedback neural networks and its most common archi-

tectures. The forth chapter(4), network training, goes through the gradient descent algorithm, its different variants and extensions but also covers the properties of the neural network's hyperparameters. Finally, the fifth chapter(5), experiments, demonstrates the comparison between Tree-based GRU and Tree-based LSTM, it also describes the gradient descent variants and the hyperparameters that were selected for the training of the model.

1.3 Background

This section provides the essential background on sequences, syntactic structures, and neural networks in order for everybody to be able to read this report.

1.3.1 Sentiment Classification

Sentiment analysis/classification [3] (also known as opinion mining) is the classification on whether a piece of text is positive, negative or neutral using NLP, statistics, or machine learning methods.

1.3.2 Sequences

A sequence is a string of objects.Each sequence is a row of item sets.The individual elements in a sequence are also called terms. In the case of the word sequences, a word corresponds to an item. In the case of the health care data, a test value is an item. Treating data as sequential (when they are sequences) improve the prediction accuracy of the classifiers. [4]

1.3.3 Syntactic Structure

The sentences can be described with two ways for a machine to make sense out of them, the first one is by breaking up the sentence to phrases(constituents), which is known as constituent structure, and the second is by connecting the words with links, which is known as dependency structure. Those structures are constructed by parsing algorithms. The parsers for constructing a constituent tree are called phrase structured parsers and the parsers for dependency structures are called dependency parsers.

Constituency Structure

The phrase structure was introduced by Noam Chomsky, idea behind constituency phrase structure is to organize the words into nested constituents[5] [6](Figure 1.1). Meaning that each nested constituent (word phrase) is a word unit. There has been different criteria for determining the constituents. The most popular is the one that claims that a constituent behaves as a unit no matter the place that is located in the sentence. Regarding to the construction of this structure, one great parser is the constituent parser from Zhu[7].

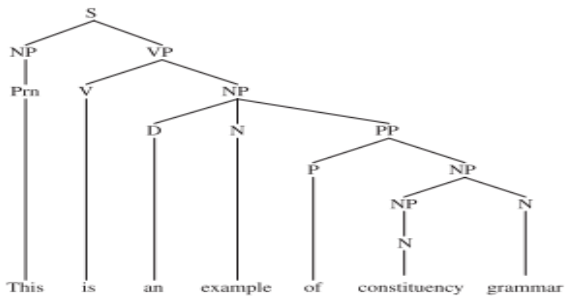


Figure 1.1: Constituency Tree

Dependency Structure

The idea behind dependency structure is having words connected with a dependency relation (Figure 1.2),where one of them is the head and the other is the dependent and there is a link connecting them. In more detail, the dependent is the modifier, object, or complement while the head determines the behavior of the pair. The dependent requires the presence of the head; the head on the other hand doesn't require the presence of the dependent. [8]. In general, the dependency structure is a tree with the main verb as its root (head of the whole structure). It is worth mentioning that a dependency structure can be constructed from constituent trees as well [9]. The idea behind dependency structure is to directly show for the words of a sentence which are the words depend on (modify or are arguments of) which other words. [10]

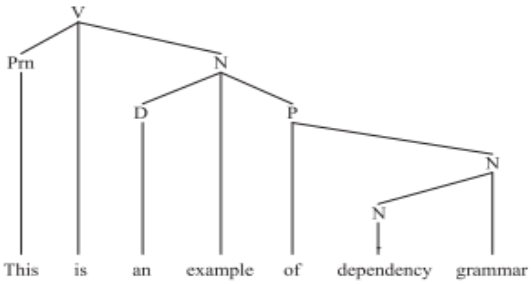


Figure 1.2: Dependency Tree

1.3.4 Neural Networks

Neural networks are models of computation that were inspired by the way (we assume) our brain works [11], [12],[13]. The structure of artificial neural networks (ANN) is a network of small processing units (neurons) that are connected with weighted joints. Over the years many variants of ANN has been proposed. One important distinction is the way they are connected,with cycles or without. The former case of neural networks are called feedback or recursive neural networks and will be examined at chapter 3, the latter case of neural networks are the Feedforward networks that will be examined at the next subsection.

Feedforward Networks

Given the absence of cycles, all nodes can be arranged into layers, and the outputs in each layer can be calculated given the outputs from the lower layers. The input i to a feedforward network is provided by setting the values of the lowest layer. Each higher layer is then successively computed until output is generated at the output layer o (Figure 1.3).

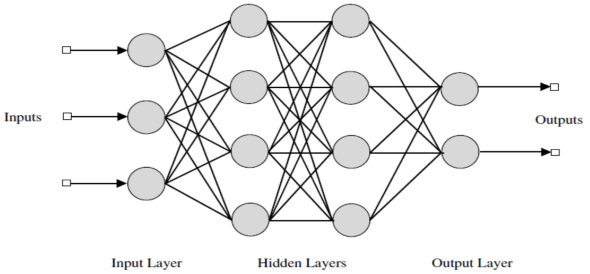


Figure 1.3: Feed-forward neural network

Let w_{jk}^l be the weight for the connection from the k^{th} neuron in the layer l -1 to the j^{th} neuron in the l^{th} layer, b_j^l the bias of node j at layer l and α_j^l for the activation of neuron j at l^{th} layer. The equation below is using the sigmoid function.

$$\alpha_j^l = \sigma(\sum_{limit=k} w_{jk}^l \alpha_k^{l-1} + b_j^l) \tag{1.1}$$

Having the equation above in mind, we can rewrite it in a more compact and vectorized form:

$$\alpha^l = \sigma(w^l \alpha^{l-1} + b^l) \tag{1.2}$$

Let $z^l = \sum_{limit=k} w_{jk}^l \alpha_k^{l-1} + b_j^l$ be the weighted input to the neurons in layer l.

$$\alpha^l = \sigma(z^l) \tag{1.3}$$

The most popular choices of activation function are the hyperbolic tangent (1.4), sigmoid (1.5), rectified linear unit(1.6), softmax (generalization of sigmoid for K classes)(1.7) .

$$tanh(x) = \frac{e^{2x}1}{e^{2x}+1} \tag{1.4}$$

$$\sigma(x) = \frac{1}{1+e^x} \tag{1.5}$$

$$f(x) = max(0, x) \tag{1.6}$$

$$softmax(x)_j = \frac{e^{xj}}{\sum_{k=1}^{K} e^{xk}}, forj = 1, ..., K \tag{1.7}$$

The most popular FNNs are perceptrons[12], Kohonen maps[14] and Hopfield nets[15] and multilayer perceptron (MLP)[13],[16], [17].

Backpropagation

The most successful algorithm for training neural networks is backpropagation, it was introduced for this purpose by Rumelhart, Hinton, Williams[13] and some alterations were suggested by Zipser[18], and Werbos [16]. Backpropagation uses the chain rule to calculate the derivative of the loss function L with respect to each parameter(weights and biases) in the network. The weights are then adjusted by gradient descent algorithm (which we will go into detail at chapter 4). While it is not certain that backpropagation will reach a global minimum (unless the loss surface is convex[1]) ,many researchers have worked on heuristic pre-training and optimization techniques that make them practically good enough for supervised learning tasks.

To calculate the gradient in a feedforward neural network, backpropagation proceeds as follows. First,proceeds to the forward pass, an example is propagated forward through the network to produce a value α_j^l, at each node j at layer l and outputs α^L at the output layer L. Then, a loss function value $\mathrm{L}(\alpha_k^L, y_k)$ is computed at each output node k. Subsequently, for each output node j, we calculate the error where the first expression $\frac{\theta L(\alpha_j^L, y_j)}{\theta \alpha_j^L}$ corresponds to the rate of change in respect to the output neuron j, and the second term measures how fast the activation function σ is changing at z_j^L :

$$\delta_j^L = \frac{\theta L(\alpha_j^L, y_j)}{\theta \alpha_j^L} \sigma'(z_j^L) \tag{1.8}$$

The equation above could be written in a more compact and matrix-based form, where $\nabla_\alpha L$ corresponds to the he rate of change of L with respect to the output activations, as:

$$\delta^L = \nabla_\alpha L \odot \sigma'(z^L) \tag{1.9}$$

Having computed the error of the output layer, we go to compute the error of the prior layer. The equation for computing the layer before is:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{1.10}$$

By combining the equations 1.9 and 1.10, we can compute the error at any layer. The backpropagation algorithm starts from the output layer L calculating the δ^L with equation 1.9 and moves to the previous layers with equation 1.10.

[1]convex surface:when the local optima is equal to the global optima

The equation for the rate of change of the cost in respect to the bias of node j in layer l is:

$$\frac{\theta L}{\theta b_j^l} = \delta_j^l \tag{1.11}$$

A more clean and vectorized form of the equation above can be written as:

$$\frac{\theta L}{\theta b} = \delta \tag{1.12}$$

The equation for the rate of change of the cost in respect to the weight that connects the node k and node j in layer l is:

$$\frac{\theta L}{\theta w_{jk}^l} = \alpha_k^{l-1} \delta_j^l \tag{1.13}$$

where α is the activation of the neuron input to the weight w and δ is the error of the neuron output from the weight w.

$$\frac{\theta L}{\theta w} = \alpha^{IN} \delta^{OUT} \tag{1.14}$$

Chapter 2

Word Embeddings

Word embedding is a representation of a word in vector space where semantically similar words are mapped to nearby points. Word embeddings can be trained and used to derive similarities between words. They are an arrangement of numbers representing the semantic and syntactic information of words in a format that computers can understand. For many years, NLP systems and techniques would represent meaning of words using WordNet (George A. Miller, Princeton University, 1985) which is basically a very large graph that defines different relationships between words. In vector space terms, every word is a vector with one 1 and a lot of zeros (vocabulary size -1). This is a so called one-hot that describes words in the simplest way. However, this discrete representation had many issues, such as missing nuances, missing new words, the requirement of human labor to create and adapt, it was hard to compute accurate word similarity and most importantly when the vocabulary is large, the vector represantation is gigantic.

The new approach of representing words was inspired by the quote "You shall know a word by the company it keeps" [19]. Instead of representing a word by its own index, represent a word by means of its words. This approach of representing words is by creating a dense embedding vector (word embedding). The word embeddings are derived by Vector Space Models (VSM) [20] which are divided in two categories which have been critically compared by Baron [21]. The first type of models is the count-based method (aka full document method) that compute the statistics of how often some words co-occur with its neighbor words in a large text corpus and then map those statistics to a low dimensional, dense vector. Some worth mentioning examples of this methodology are Hyperspace Analogue to Language [22], COALS method [23], Hellinger PCA [24]. The most pop-

ular model of the count-based method is the Latent Semantic Analysis [25] which we will cover at the section (2.1). The second type of VSM models are the predictive models which try to predict directly the word from its neighbors in terms of learned low dimensional, dense embedding vectors. Some models worth mentioning of this category are Semantic Role Labeling(SRL) [26], Mnih and Kavukcuoglu vLBL and ivLBL[27],[28],Levy [29] proposed explicit word embeddings based on a PPMI metric. At sections 2.2 and 2.3 we will cover the most popular models of the VSM predictive models , named Word2Vec and GloVe. The 2.1 and 2.2 are covered just to demonstrate way that GloVe 2.3 was conceived, therefore they are covered with not much details.

The new approach of word representation tackles the problems of high dimensionality, the scalability of the vocabulary and the semantic relatedness of the words.

2.1 Latent Semantic Analysis

Latent semantic analysis (LSA) is a methodology in natural language processing of analyzing relationships between a set of documents and the terms they contain by producing a set of concepts related to the documents and terms. LSA assumes that words that are close in meaning will occur in similar pieces of text. The term-document matrix that is created, is quite sparse. Therefore a mathematical technique called singular value decomposition (SVD) is used to reduce the number of rows while preserving the similarity structure among columns. Singular Value Decomposition [30] is a method for identifying and ordering the dimensions of the observation that exhibit the most variation. Once we have found where the most variation lies, we can find the best approximation of the original observation using fewer dimensions. Therefore, it can be used for dimensionality reduction tasks.

Let X be a matrix with m terms and n documents where element (i, j) describes the occurrence of term i in document j co-occurrence matrix. According to SVD, there is a decomposition of X so that U and V are orthogonal matrices and Σ is a diagonal matrix.The values $s_1, \ldots, s_l$ are called the singular values, and $u_1, \ldots, u_l$ and $v_1, \ldots, v_l$ the left and right singular vectors.

$$\underset{}{\overset{X}{\begin{bmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{bmatrix}}} = \underset{}{\overset{U}{\begin{bmatrix} u_{11} & \cdots & u_{1r} \\ \vdots & \ddots & \vdots \\ u_{m1} & \cdots & u_{mr} \end{bmatrix}}} \underset{}{\overset{\Sigma}{\begin{bmatrix} s_{11} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & s_{rr} \end{bmatrix}}} \underset{}{\overset{V^T}{\begin{bmatrix} u_{11} & \cdots & u_{1r} \\ \vdots & \ddots & \vdots \\ u_{n1} & \cdots & u_{rn} \end{bmatrix}}}$$

Moreover the The matrix product XX^T gives us the the correlation between the terms over the set of documents and the X^TX gives us the correlation between the documents over the set of terms.

$$\begin{aligned} XX^T &= (U\Sigma V^T)(U\Sigma V^T)^T = (U\Sigma V^T)(V^{T^T}\Sigma^T U^T) = U\Sigma V^T V\Sigma^T U^T = U\Sigma\Sigma^T U^T = U\Sigma^2 U^T \\ X^TX &= (U\Sigma V^T)^T(U\Sigma V^T) = (V^{T^T}\Sigma^T U^T)(U\Sigma V^T) = V\Sigma^T U^T U\Sigma V^T = V\Sigma^T\Sigma V^T = V\Sigma^2 V^T \end{aligned}$$

Since $\Sigma\Sigma^T$ and $\Sigma^T\Sigma$ are diagonal we can safely conclude that U are the eigenvectors of XX^T, V are the eigenvectors of X^TX and both products have the same eigenvalues, given by the entries of $\Sigma\Sigma^T$ or $\Sigma^T\Sigma$. From Frobenius norm[1][31] we can derive that by taking the k largest singular values (express the importance of every word), and their corresponding singular vectors, we get the rank k approximation to X with the lowest error. The word vectors of the corpus will be the k columns of the matrix .

$$\underset{}{\overset{X}{\begin{bmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{bmatrix}}} \approx \underset{}{\overset{\hat{U}}{\begin{bmatrix} u_{11} & \cdots & u_{1k} \\ \vdots & \ddots & \vdots \\ u_{m1} & \cdots & u_{mk} \end{bmatrix}}} \underset{}{\overset{\hat{S}}{\begin{bmatrix} s_{11} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & s_{kk} \end{bmatrix}}} \underset{}{\overset{\hat{V^T}}{\begin{bmatrix} u_{11} & \cdots & u_{1k} \\ \vdots & \ddots & \vdots \\ u_{n1} & \cdots & x_{kn} \end{bmatrix}}}$$

While this method solves the problem of dimensionality, it underlies some problems as well. First of all, the computational cost increases quadratically as the size of the matrix increases. Moreover when new words appear SVD has to be run again from scratch. Finally, it is able to find similarities between words, but it cannot represent relationships.

2.2 Word2Vec

Word2Vec (Mikolov, 2013) is a predictive model that learns word embeddings on an online way. The main idea behind Word2Vec is to predict the surrounding words of every word in a window of length m, instead of capturing all the co-occurrence counts directly. It is simpler and faster than LSA and can easily add a new word to the vocabulary.

The objective function of predictive VSMs (aka Neural probabilis-

[1]Frobenius norm: is matrix norm of an mn matrix A defined as the square root of the sum of the absolute squares of its elements

tic language models) aim to maximize the average log probability of any context word given the current center word where t is the number of tokens, m the co-occurrence window and θ all the variables we optimize using stochastic gradient descent.

$$J(\theta) = \frac{1}{T}\sum_{i=1}^{T} \sum_{-m \leq j \leq m, j \neq 0} \log p(w_{t+j}|w_t) \tag{2.1}$$

For $\mathrm{p}(\mathrm{w}_{t+j}|w_t)$:

$$p(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w=1}^{W} \exp(u_w^T v_c)} \tag{2.2}$$

where o is the output word id, c is the center word id, u is the center word vector and v is the output vector. This objective function is not scalable and it takes much time to train when the vocabulary is large.

Those two models are great at constructing word vectors, but when the vocabulary becomes too large the updates at each iteration take too much time. This problem is solved by a method called Negative Sampling [32]. This idea suggests to take random samples from the vocabulary that do not appear on the context and minimizing their probability of occurring 2.3.

$$J(\theta) = \frac{1}{T}\sum_{t=1}^{T} J_t(\theta) \tag{2.3}$$

where:

$$J_t(\theta) = \log\sigma(\mathrm{u}_o^T v_c) + \sum_{i=1}^{k} \mathbb{E}_{j \sim P(w)}[\log\sigma(\text{-}\mathrm{u}_j^T v_c)] \tag{2.4}$$

This objective function maximizes the probability of $u_o^T v_c$ co-occur and minimizes the probability the words randomly selected co-occur.

The most popular Word2Vec variants are CBOW [28], which stands for continuous bag of words, and Skip-gram [32]. CBOW (Figure**??**) predicts

the current word based on the context.More precisely, it predicts the current word based on the n neighbours that occur before and n neighbours that occur after than this word.The (window size) inputs share the weights that connect them with the hidden layer, what happens to the hidden layer, is to take the mean of the input words and it passes along to the output. Skip-gram (Figure**??**) is the opposite of CBOW, while CBOW uses the context to predict the middle words Skip-gram uses the middle word to predict the context (window size). For the most part, CBOW tends to be a useful technique for smaller datasets. On the other hand, skip-gram treats each context-target pair as a new observation, therefore it tends to perform better with larger datasets.

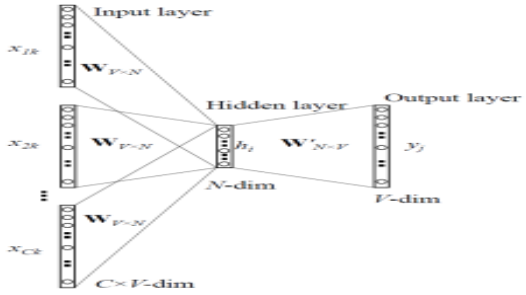


Figure 2.1: CBOW

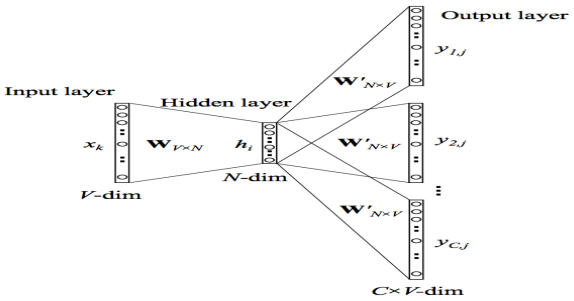


Figure 2.2: Skip-Gram

Finally according to Mikolov, Skip-gram works well with small amount of training data, and CBOW performs better in bigger amount of training data. All in all, Word2Vec models generate improved performance and are capable of capturing complex patterns beyond word similarity but they scale with corpus size and make inefficient usage of statistics.

2.3 GloVe

GloVe, which stands for global vectors, is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations are linear substructures of the word vector space.[33]

GloVe combines the advantages of SVD and Word2Vec, it trains fast, it is scalable to huge corporas and performs well even with small corpus and small vectors. The main idea is, instead of going over one window at a time, it collects the whole corpus and is trained on the non-zero entries of the matrix. The weighted least squares regression model of GloVe is presented

below 2.5,where W is the size of the vocabulary, $P_{i,j}$ is the probability that word j appear in the context of word i ,$f()$ is the weighting function, u_i^T is the center vector of word i and v_j is the neighbor word j.

$$J(\theta) = \frac{1}{2} \sum_{i,j=1}^{W} f(P_{ij})(u_i^T v_j - \log P_{ij})^2 \tag{2.5}$$

The weighting function (Figure 2.3) should obey the following properties:

- $f(0) = 0$, If f is viewed as a continuous function, it should vanish as $x \to 0$ fast enough that the $\lim_{x \to 0} f(x) log^2 x$ is finite.
- $f(x)$ should be non-decreasing so that rare co-occurrences are not overweighted.
- $f(x)$ should be relatively small for large values of x, so that frequent co-occurrences are not overweighted.
- $f(x) = \begin{cases} (x/x_{max})^a & if \quad x<x_{max} \\ 1 & otherwise \end{cases}$

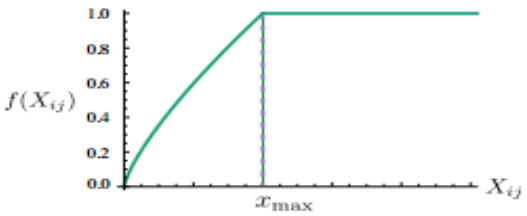


Figure 2.3: Weighting function f with = 3/4.

The goal of the objective funtion 2.5 is to learn word vectors such that their inner product $(u_i^T v_j)$, which corresponding to the prediction of those words co-occurring, equals to the logarithm of the word's probability of co-occurrence $(\log P_{ij})$. The reason behind using the logarithms of word's probability is the fact that the logarithm of a ratio equals the difference of logarithms, so it makes it feasible to illustrate probability of co-occurrence of two words as vector differences in the word vector space. For this reason, the resulting word vectors perform very well on word analogy tasks. (e.g. King - Man $\approx$ Queen - Woman)

Finally, the GloVe model compared to the two most popular Word2Vec models ,Skip-Gram (Figure 2.4) and CBOW (Figure 2.5),the plots illustrate the overall performance on the analogy task as a function of training time. Under the very same conditions GloVe outperforms word2vec. It trains faster and better irrespective of speed.

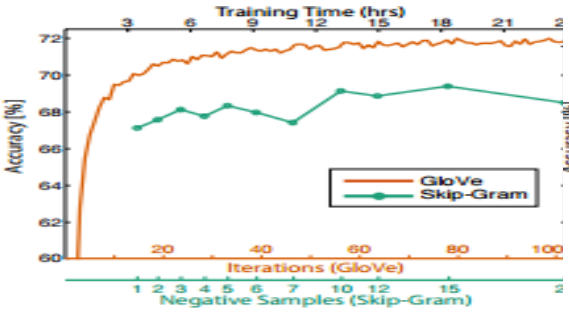


Figure 2.4: GloVe vs Skip-gram

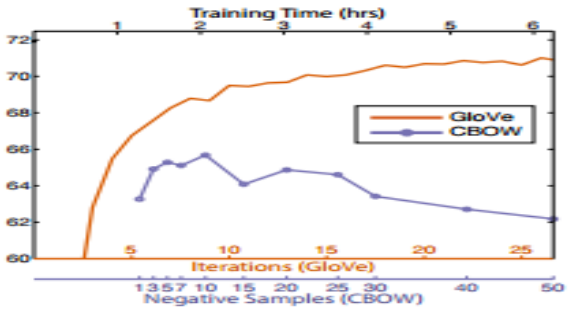


Figure 2.5: GloVe vs CBOW

Chapter 3

Feedback Neural Networks

This chapter aims to provide the reader with the background of feedback neural networks. In more detail, at section 3.1 we will review recurrent neural networks (simplistic subset of recursive neural networks) and at section 3.2 we will review recursive neural networks (also known as Tree-based recurrent neural networks).

3.1 Recurrent Neural Networks

A special case of recursive neural networks are the Recurrent Neural Networks (Figure 3.1) whose structure corresponds to a linear chain. The idea behind Recurrent Neural Networks is to make use of the sequential nature of the data. While in traditional neural networks we assume that input are independent to each other, in RNN we consider the importance of time. The reason that they are selected for sequential problems, is that they capture the information from each time-step, in practice though, they capture only a few steps but we will see the RNN variants that tackle this problem at subsections 3.1.2 and 3.1.3. Recurrent neural networks, has been under extensive research for quite some time now, so many variants of RNN have been proposed, such as Elman networks[34], Jordan networks[35], time delay neural networks[36], Long-Short Term Memory networks[37], echo state networks[38], Gated Recurrent Units[39]. In this section we will present Elman's networks(3.1.1, aka Simple recurrent networks), Long-Short Term networks(3.1.2), and Gated Recurrent Units(3.1.3).

3.1.1 Simple Recurrent Neural Networks

Forward Pass

The forward pass of an RNN is the same as that of an FNN, apart from the fact that the activations arrive at the hidden layer from the input layer (from the same time-step) and from the hidden layer activations one time step before.

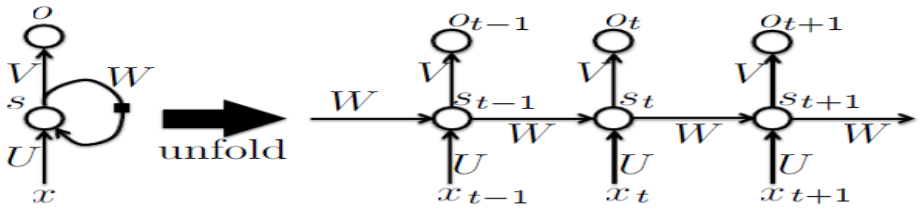


Figure 3.1: Recurrent Neural Network

In more detail,(See Figure 3.1) x_t is the input at time step t, h_t is the hidden state at time step t 3.1(this is what differentiates FNN from RNN, h_t is the network's memory) ,and o_t is the output at time step t 3.2 .

$$h_t = \tanh(Wx_t + Uh_{t-1}) \tag{3.1}$$

$$o_t = softmax(Vh_t) \tag{3.2}$$

Note that this requires initial values h_0 to be chosen , which correspond to the network's state before having any inputs. The initialization of the previous state can set the values to zero or initialize with a non zero initial values that are adjusted over the training process [40] which in some cases can improve the robustness and stability of the network.

Backward Pass

Regarding the RNN's training, the Backpropagation Through Time algorithm is applied [18][16]. BPTT is very similar to the Backprogation (1.3.4) algorithm. The reason BPTT is used, is because it is more efficient in computating time. Moreover it is an algorithm under active research, one example

is the new state of art way of computing BPTT has been published [41] which uses dynamic programming to balance a trade-off between caching of intermediate results and recomputation.

Just like Backpropagation (1.3.4) BPTT is consisted of repeated application of the chain rule. The only difference is that the objective function depends on the activation of the hidden layer not only because of its influence at output layer (o_t), but also on the hidden layer at the next time-step (h_{t+1} , which means that it affects the next steps to come as well). It is important to notice that the weights between time steps share the same values and that we sum over the whole sequence to get the derivatives with respect to each of the network weights. Mathematically:

$$\delta_{t,h} = \theta'(v_{t,h})(\sum_{k=1}^{K} \delta_{t,k} u_{hk} + \sum_{l=1}^{H} \delta_{t+1,l} u_{hl}) \tag{3.3}$$

where,

$$\delta_{t,j} = \frac{\theta O}{\theta v_{t,j}} \tag{3.4}$$

The complete sequence of δ terms can be calculated by starting at last time step and recursively applying 3.3, decrementing t at each step. Finally to take the derivatives with respect to the network's weights, we sum over the whole sequence (since a single training instance is the full sequence).

$$\frac{\theta O}{\theta u_{ij}} = \sum_{t=1}^{T} \frac{\theta O}{\theta v_{tj}} \frac{\theta v_{tj}}{\theta u_{ij}} = \sum_{t=1}^{T} \delta_{t,j} b_{t,i} \tag{3.5}$$

where, $b_{t,i}$ is the activation of unit i at time t

While RNNs work well with sequences, they have a major drawback which is the vanishing/exploding gradient problem[42],[43]. One approach to deal with it is called the truncated BPTT [42] which is to set a limit on the time steps that you consider. Other approaches have been taken in order to tackle this problem .The most important were discrete error propagation [43], time delays [36], hierarchical sequence compression [44],Hessian Free Optimization [45], and Echo State Networks [46]. However, the most effective approach was a whole different architecture, the Long Short Term Memory (LSTM) architecture [37]. Moreover, RNNs have no control on "how many items/time steps to remember". The context needed for each classification task varies significantly, it may need 3 time steps (words) or 20. This problem is also solved with the LSTM architecture.

3.1.2 Long-Short Term Memory

Long-Short Term Memory (Figure 3.2) is the most popular architecture for sequential data, it is almost a subject under active research and constant improvement. LSTM networks have been applied to a variety of sequence modelling and prediction tasks with state-of-art performance, notably machine translation [47],[48], speech recognition [49], image caption generation [50],and program execution [51]. In its original form, LSTM contained only input and output gates. The forget gates [52], along with additional peephole weights [53] connecting the gates to the memory cell were added later. LSTMs are designed to be able to keep the important information and forget the noise. The default behavior of LSTM is to remember long periods and after training it remembers only the important information.

Architecture

Long-Short Term Memory networks' structure is like Simple RNN's, with only difference on the hidden layer structure. While RNNs have a single layer, the LSTMs have four hidden layers. The LSTM is consisted of cells and gates, cells contain the information and gates regulate "how much" of information to let go, gates are composed of a sigmoid layer that outputs an interval from 0 to 1 where 0 stands for "pass no information" and 1 for "pass everything". In more detail, it is consisted of the forget gate (f_t) which decides how much information (current and past) to pass to the network, the input gate (i_t) which decides what information to store at the cell, the cell state(C_t) which is the updated state, and finally the output gate.

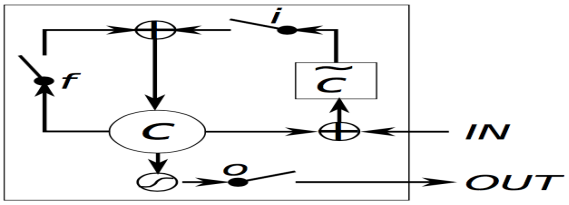


Figure 3.2: Long-Short Term Memory

Forward Pass

The first step is to decide what to input to the network, here is where forget gate comes in (3.7) where takes as input the previous state (h_{t-1}) and the current input (x_t) and it outputs an interval between 0 to 1 to the

candidate cell state($\tilde{C}_t$)(3.8). Then we decide what values to update (3.9) at the input gate(i_t). After having computed candidate cell and the input gate we combine them with the previous cell state(C_{t-1}) to derive the current cell state (C_t)(3.10). Finally we feed the cell state to the output gate in order to decide how much of the cell state to output(3.11) as well as to a tanh layer so to squash the values between -1 and 1.

$$i_t = \sigma(W^{(i)}x_t + U^{(i)}h_{t-1}) \tag{3.6}$$

$$f_t = \sigma(W^{(f)}x_t + U^{(f)}h_{t-1}) \tag{3.7}$$

$$o_t = \sigma(W^{(o)}x_t + U^{(o)}h_{t-1}) \tag{3.8}$$

$$\tilde{C}_t = \tanh(W^{(C)}x_t + U^{(C)}h_{t-1}) \tag{3.9}$$

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \tag{3.10}$$

$$h_t = o_t \odot tanh(C_t) \tag{3.11}$$

Backward Pass

The original LSTM training algorithm [37] used an approximate error gradient calculated with a combination of Real Time Recurrent Learning [54] and Backpropagation Through Time (BPTT) [18].The BPTT part was truncated after one time-step, since memory blocks would deal with the longer dependencies. The truncating method has the benefit of making the algorithm online, meaning that weight updates can be made after every time-step. The best performing algorithm is BPTT from Alex Graves[**?**] , for further details look at appendix A.

3.1.3 Gated Recurrent Unit

Gated recurrent unit (Figure 3.3) is another variant of recurrent units proposed by KyungHyun Cho[2]. It is closely related Long-Short Term Memory. The GRU also controls the flow of information like the LSTM, but without using a memory unit. It exposes the full hidden content without any control.

Architecture

A GRU has two gates, a reset gate r, and an update gate z. The reset gate indicates how to combine the new input with the previous memory. The update gate defines how much of the previous state to keep.The basic idea of using a gating mechanism to learn long-term dependencies is the same as in a LSTM, but there are a few differences in terms of architecture. First of all, it doesn't have an output gate so it has fewer parameters (two gates, instead of three). Secondly the input and forget gates are substituted by an update gate z and the reset gate r is applied directly to the previous hidden state. Thus, the responsibility of the reset gate in a LSTM is really split up into both r and z. Finally we don't apply a second nonlinearity when we compute the output.

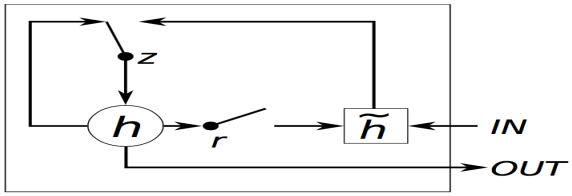


Figure 3.3: Gated Recurrent Unit

Forward Pass

The first step is to determine the input values, this is update gate's task it will decide how much of the previous hidden state and how much of the candidate hidden state combines to get the new hidden state (3.12). After the update gate, the reset gate while it has the exact same functional form (3.13) as the update gate and all the weights are at the same size, what makes it different is its position at the model. The reset gate is multiplied by the previous hidden state it controls how much from the previous hidden state we will consider when we create the new candidate hidden state. In other words, it has the ability to reset the hidden state , if we set the reset gate to 0 from (3.14) we start over from a new sequence as if h_t is the beginning of a new sequence. However this is not the full picture since $\tilde{h}_t$ is just a candidate for the hidden state, the actual hidden state will be a combination of previous hidden state h_{t-1} and the candidate hidden state $\tilde{h}_t$ controlled by the update gate z_t (3.15) .

$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1}) \tag{3.12}$$

$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1}) \tag{3.13}$$

$$\tilde{h}_t = \tanh(W^{(h)}x_t + U^{(h)}(h_{t-1} \odot r)) \tag{3.14}$$

$$h_t = (1 - z) \odot \tilde{h}_t + z \odot h_{t-1} \tag{3.15}$$

Backward Pass

Gated Recurrent Units also use the BPTT algorithm in order to be trained. We will derive the gradients for E (error),W,U and by hand using the chain rule, for further details look at appendix B

3.2 Recursive Neural Networks

Recursive Neural Networks (RNNs) [55], [56], [57], [58], [59] are perfect for settings that have nested hierarchy and an intrinsic recursive structure [60]. The syntactic rules of language are highly recursive, therefore we use that recursive structure with a model that complies with that property. It is important to notice that RNN don't comprehend sentences as sequences but as hierarchies which makes them ideal for semantic representation tasks (paraphrase detection [55], relation classification, sentiment analysis, phrase similarity) but they can't predict future items from a sequence(next word from a given sentence),something that Recurrent Neural Networks are very good at due to their linear structure. Recurrent Neural Networks represent sentences as parse trees(Figure 3.4).

What RNNs are very good at, is handling negation. Due to their hierarchical structure, when negation is spotted, the meaning is just being reversed. You have a label at every node of the tree, and the leaves of the tree represent words.

Moreover another reason that RNN are so popular for natural language processing tasks, is that the input sequence length (sentence in our case) is not a restriction, it can take inputs of arbitrary lengths. The latter benefit is accomplished by making the input vector of sentence a predefined

size no matter the length of the sentence. ([61],[62], [63]. Essentially what RNNs do is to merge the semantic understanding[1] of the words, then the grammatic understanding[2] of the phrase or sentence, which results to a parse tree representation of a phrase or a sentence. Having understood the words, and knowing the way words are put together we can retrieve the meaning of the sentence. Even though grammatical understanding is an assumption and it is not proven that it improves the accuracy, it is still under debate but we will assume that it helps the model.

In short, it extracts from the sentence the syntactic structure, which indicates the relationship between phrases, and it identifies the meaningful phrases within the sentences and the relationship between them. In order to extract the vector representation of the sentence, the idea is to recursively merge pairs of representations of smaller segments to get representation that covers bigger senteces.

The Recursive Neural Networks are trained with the Backpropagation Through Structure algorithm[57] which is very similar to the standard Backpropagation, we use the 1.10 and the 1.14, that was discussed at subsection 1.3.4 with three minor differences. Firstly we sum up the derivatives of W from all the nodes, secondly we split the derivatives at each node and finally we add different error messages from parent node and self node.

Finally, it is assumed that the tree structured is given, which indicates that some preprocessing is required if it is not given. In our experiment we will use the Stanford Sentiment Treebank(SST) that was trained with the Stanford Parser[64] (which is similar to max-margin parsing [65]) to derive the tree structure for every sentence. We will not go through the way that the sentence trees were constructed because we will add unnecessary complexity that is beyond the scope of this report, but at a high level explanation the parser that is used, have a loss term that penalizes the not plausible phrases.

3.2.1 Simple Recursive Neural Network

This model (Figure 3.5) is the standard recursive neural network. The first step to take is to take a sentence parse tree and the sentence word vectors and begin from the bottom leaves to the top root of the tree. The mathematical formula to merge those two vectors (aka children) and create a new

[1] semantic understanding: understanding of the meaning of a sentence, represent accurately the phrase as a vector in a structured semantic space

[2] grammatical understanding: it is identified the underlying grammatical structure of the sentence

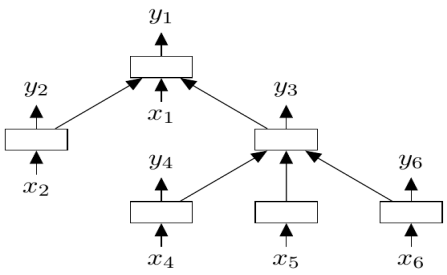


Figure 3.4: Recursive Neural Network

"word phrase" vector (parent) can be illustrated below (3.16). The h vector now represent the "this assignment" phrase. Having computed the vector representations of the sentences, we compute a **s** score 3.17 which represents the quality of the merge and decides which pair of representations to merge first. In oder to derive some meaning of the word vector, we feed it to a softmax layer (3.18) to compute the score over a set of sentiment classes, a discrete set of known classes that represent some meaning. This process happens till the model reach the root of the tree. Moreover it is important to note that the W parameters is the same for all the nodes of the trees. It is obvious that this is quite a naive approach and linguistic complexity is higher than that. It is too much to ask from a simple function like this to capture the language complexity.

$$h_1 = \tanh(W \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} + b) \tag{3.16}$$

$$s_1 = W^{score} p \tag{3.17}$$

$$\hat{y} = softmax(W h_1 + b) \tag{3.18}$$

3.2.2 Syntactically Untied SU-RNN

One extension to the Simple RNN, the Syntactically Untied RNN model [60](Figure 3.6) was introduced to solve the problem mentioned at the previous subsection. What this model does, is to have unique weight matrices

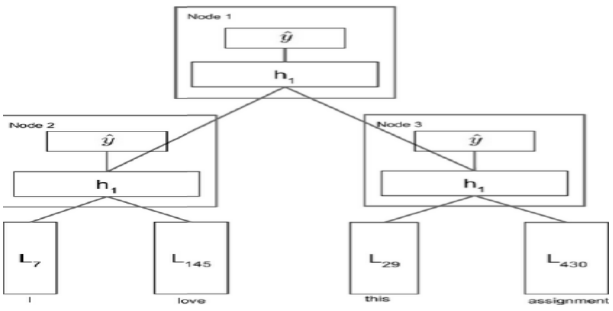


Figure 3.5: Simple Recursive Neural Network

for every syntactic category. The syntactic categories are identified from the parser that determined the structure of the tree. This has proven to increases the weight matrices to learn and outperforms the methods that were mentioned till that point,but the performance boost we gained is not significant.

One impressive accomplishment of this model is that the trained weight matrices are capable of learning the semantics of the phrases. For example a determiner followed by a noun phrase (e.g. "an elephant") emphasizes more on the noun phrase than on the determiner. The architecture of the SU-RNN model compared to the Simple RNN is illustrated at Figure 3.6.

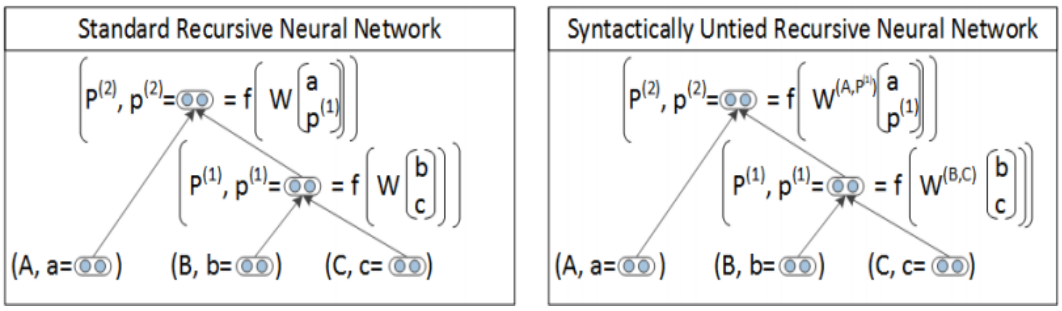


Figure 3.6: Syntactically Untied Recursive Neural Network

3.2.3 Matrix-Vector Recursive Neural Networks

Another alteration of Recursive Neural networks is the Matrix-Vector Recursive Neural Networks [66] (Figure 3.7) which improves the semantic representation of the sentences. The major difference is that not only we include a word vector (d-dimensional), but also a word matrix (dXd)(Figure 3.7).

$$h_1 = \tanh(W \begin{bmatrix} C_2 c_1 \\ C_1 c_2 \end{bmatrix} + b) \tag{3.19}$$

This approach not only represents the meaning of each word but also the effect that it has on the neighboring words. Suppose we feed two words to the model, a and b, the parent vector is the concatenation of the word vector of the former multiplied with the word matrix of the latter and the word vector of the later is multiplied by the word matrix of the former (Ab and Ba). In the figure's example, the word matrix of "very" could also be the identity[3]multiplied by a scalar (above one) which indicates the impact it has to the word "bad".

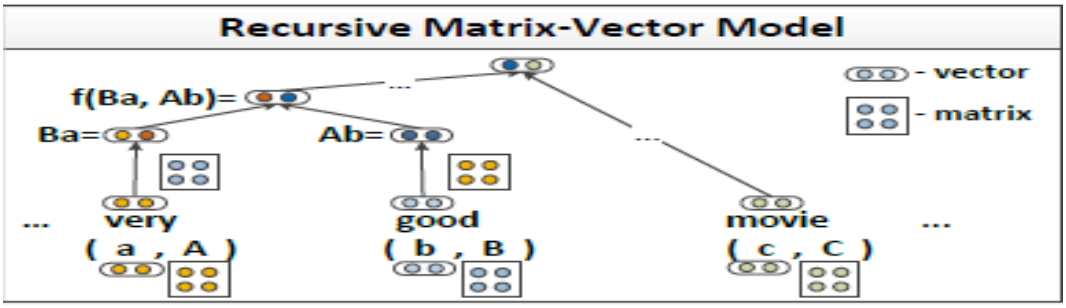


Figure 3.7: Matrix-Vector Recursive Neural Networks

Despite the fact that this is the most expressive model we have explored till now, it is still not good enough. It fails to capture the semantics of some relations. There have been observed three types of errors[60]. The first type (Figure 3.8) is the negated positives, this case occurs when something is classified as positive but one word turns it negative, the model can not capture the importance of that one word strong enough to flip the sentiment of the entire sentence.

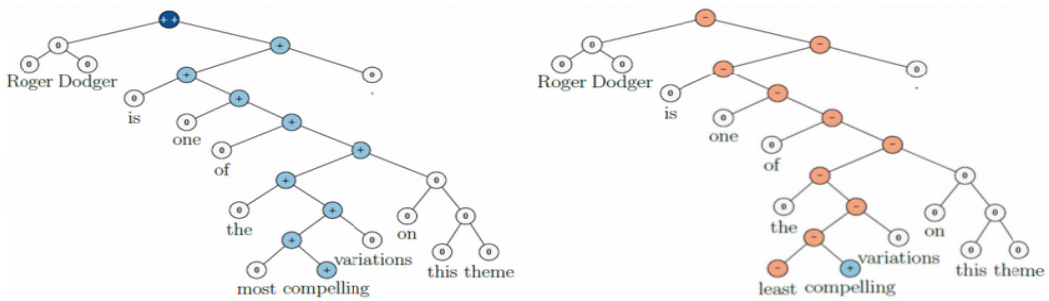


Figure 3.8: Negated positives

[3]identity matrix: square matrix with ones on the main diagonal and zeros elsewhere

The second type (Figure 3.9) is the negated negative, where we say something is not bad. The MVRNN can not recognize that the word not because it turns sentiment from negative to neutral.

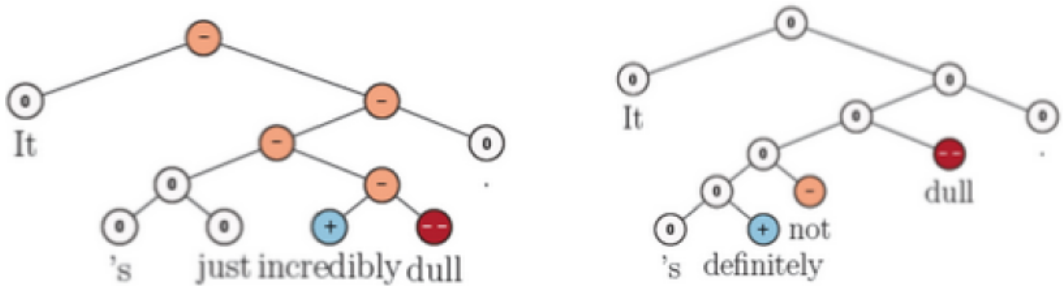


Figure 3.9: Negated Negative

The final type of errors (Figure 3.10) we observe is the X but Y conjunction. In our example, X is negative but the Y is positive and the sentiment is positive. The MV-RNNs have some issues with such cases.

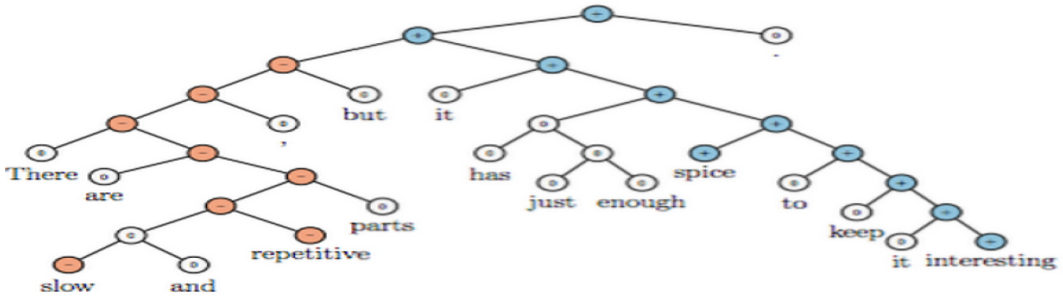


Figure 3.10: X but Y conjunction

Thus, we must look for an even more expressive composition algorithm that will be able to fully capture these types of high level compositions.

3.2.4 Recursive Neural Tensor Network

The third Recursive Neural Network variant that will be covered is the Recursive Neural Tensor Network(Figure 3.11). RNTN was conceived by Richard Socher [60] in order to solve the three types of errors we left of with at the subsection 3.2.3. Moreover it is famously quite sucessful for dealing with double negations. The Recursive Neural Tensor Network gets rid of the concept of a word matrix as well as the affine transformation [4] pre-tanh /σ concept that we saw before. To combine two word vectors or phrase vectors,

[4]affine transformation: is a transformation composed of a linear function+ a constant

we again concatenate them to form a vector $\in {}^{2d}$ but instead of putting it through an affine function then a nonlinear, we put it through a quadratic first, then a nonlinear, such as:

$$h(1) = \tanh(\mathrm{x}^T V x + W x) \tag{3.20}$$

Where V is a 3rd order tensor $\in^{2d \times 2d \times d}$. The quadratic shows that we can indeed allow the multiplicative type of interaction between the word vectors without needing to maintain and learn word matrices.
Figure 3.11: One slice of a RNTN. Note there would be d of these slices.

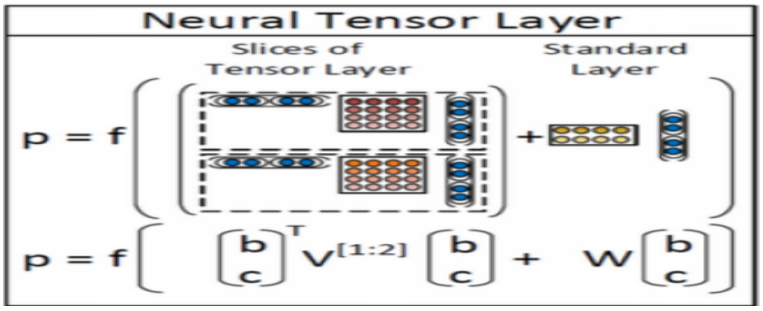


Figure 3.11: Recursive Neural Tensor Network

A major problem of the models we covered before is their inability to handle negation[67]. The table 3.1 below shows how the RNTN handles negations.

Table 3.1: Negations

Model	Negated Positive	Negated Negative
RNN	33.3	45.5
MV-RNN	52.4	54.6
RNTN	71.4	81.8

3.2.5 Tree-Based Long-Short Term Memory Networks

Tree-Based LSTM (Figure 3.12) was recently conceived by Kai Sheng Tai [1]. This is a hybrid model that combines LSTMs and Recursive neural network. It is important to notice that LSTMs were used for linear chain structured recursive neural networks (recurrent neural networks). The main difference that this model has with the standard LSTMs is that it is required the average of the child vectors and a special forget gate for each child. The

idea behind this architecture is mostly to handle negation, by keeping in memory the semantically important words and forgetting the non significant. This process happens as the model goes through the tree structure. The Figure3.12 illustrates how the new memory cell c_1 and hidden state h_1 are composed with two children.

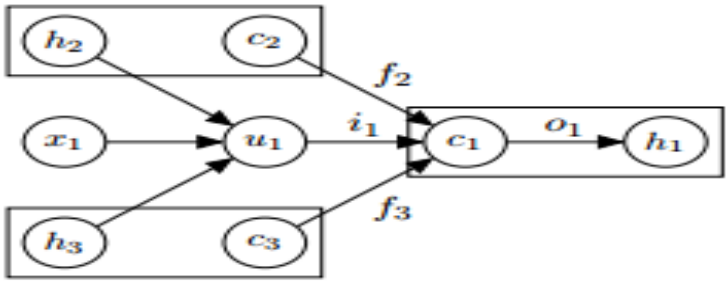


Figure 3.12: Tree-Based LSTM Memory Cell Composition

The Tree-LSTM behaves very similar to the standard LSTM, it takes as input vector x_j with only difference that the input vector depends on the tree structure (1.3.3). If the tree is a constituency tree, the leaf nodes take the corresponding word vectors as input, if the tree is a dependency tree each node in the tree takes the vector corresponding to the head word as input. The two extensions that Tai(2015) proposed are the Child Tree-LSTM and N-ary Tree-LSTM.

Child-Sum Tree-LSTM

Child-Sum Tree-LSTM unit conditions its components on the sum of child hidden states h_k, this model performs well with high branching factor tree structures or with structures that its children are not ordered. Dependency trees is a good choice of structure for that model since the number of dependents of a head can be quite variant. Let $C(j)$ the set of children of node j and $k \in C(j)$, the equations of the model are the following:

$$\tilde{h}_j = \sum_k h_k \tag{3.21}$$

$$i_j = \sigma(W^i x_j + U^i \tilde{h}_j + b^i) \tag{3.22}$$

$$f_{jk} = \sigma(W^f x_j + U^f h_k + b^f) \tag{3.23}$$

$$o_j = \sigma(W^o x_j + U^o \tilde{h}_j + b^o) \tag{3.24}$$

$$\tilde{C}_j = \tanh(W^u x_j + U^u \tilde{h}_j + b^u) \tag{3.25}$$

$$C_j = i_j \odot \tilde{C}_j + \sum_{k \in C(j)} f_{jk} \odot C_k \tag{3.26}$$

$$h_j = o_j \odot \tanh(C_j) \tag{3.27}$$

The matrix parameters on the equations above can be interpreted as encoding correlations between the input x_j, the hidden states h_k of the children and the component vectors of the Tree-LSTM. A natural extension of this model is the Dependency Tree-LSTM which is the application of the Child-Sum Tree-LSTM to a dependency tree. This model has proven not to perform that well, I assume that this can be a result of its simplistic architecture and its lack of use of the sentiment contained at the leaves.

N-ary Tree-LSTM

On the other hand, the N-ary Tree-LSTM perform well where the branching factor is less or equal to N and the children are ordered (constituent). For any j node, the hidden state is h_{jk} and its memory cell is c_{jk}. The equations of the model are the following :

$$i_j = \sigma(W^i x_j + \sum_{l=1}^{N} U_l^i h_{jl} + b^i) \tag{3.28}$$

$$f_{jk} = \sigma(W^f x_j + \sum_{l=1}^{N} U_{kl}^f h_{jl} + b^f) \tag{3.29}$$

$$o_j = \sigma(W^o x_j + \sum_{l=1}^{N} U_l^o h_{jl} + b^o) \tag{3.30}$$

$$\tilde{C}_j = \tanh(W^u x_j + \sum_{l=1}^{N} U_l^u h_{jl} + b^u) \tag{3.31}$$

$$C_j = i_j \odot \tilde{C}_j + \sum_{l=1}^{N} f_{jl} \odot c_{jl} \tag{3.32}$$

$$h_j = o_j \odot \tanh(C_j) \tag{3.33}$$

It is important to notice that the N-ary Tree-LSTM model have separate parameter matrices for each child, which results in better learning of fine-grained conditioning on the states of a unit's children than the ChildSum Tree-LSTM. It can be considered as a case of trade-off of performance and computational cost. Suppose that an example of a constituent tree that its left child of a node is a noun phrase, and the right child a verb phrase. It is beneficial for this case to magnify the verb phrase. N-ary tree is able to do that by training the U_{kl}^f parameters so that the components of f_{jl} are close to 0 ("forget") while the components of f_{j2} are close to 1 ("remember").

A natural extension to N-ary LSTM model is the Constituency Tree-LSTM which is an application of Binary Tree-LSTM. This is the model that we will compared on the later section.

3.2.6 Tree-Based Gated Recurrent Unit

Tree-Based Gated Recurrent Unit (Figure 3.13) is a model that was inspired by the Tree-Based LSTM architecture. As far as I know it hasn't formally

been published anywhere so I can't give the credentials to someone. I have developed the alteration of N-ary Tree-LSTM named N-ary Tree-GRU. The idea is almost the same with the N-ary Tree-based LSTM. However instead of having two forgetting gates, we will have two reset gates. In our experiment, the tree-structure of the inputs is contituency trees. It is important for the reader to note that both architectures (Tree-LSTM and Tree-GRU) have an affect only on the composition of the parent node. Other than that, they are like the simple recursive neural network. In more detail, the N-ary Tree-GRU takes as input word vectors and it creates the candidate hidden state $\tilde{h}_j$ with the combination of the child nodes, its child has its own reset gate. After having the candidate hidden state $\tilde{h}_j$, we calculate the actual hidden state h_j.

N-ary Tree-GRU

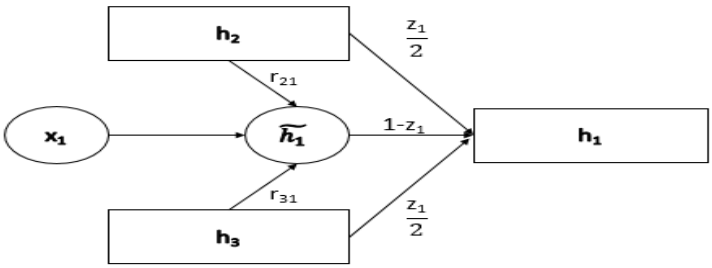


Figure 3.13: Tree-Based GRU Hidden State Composition

$$z_j = \sigma(W^z x_j + \sum_{l=1}^{N} U_l^z h_{jl} + b^z) \tag{3.34}$$

$$r_{jk} = \sigma(W^r x_j + \sum_{l=1}^{N} U_{kl}^r h_{jl} + b^r) \tag{3.35}$$

$$\tilde{h}_j = \tanh(W^h x_j + \sum_{l=1}^{N} U_l^h (h_{jl} \odot r_{jl}) + b^h) \tag{3.36}$$

$$h_j = (1 - z_j) \odot \tilde{h}_j + \sum_{l=1}^{N} \frac{z_j}{N} \odot hjk \tag{3.37}$$

Chapter 4

Neural Network Training

This chapter is devoted to the training process of a neural network which is consisted of the backpropagation and gradient descent algorithms[68]. We have discussed in great detail regarding the backpropagation algorithm at the subsection 1.3.4 and sections 3.1, 3.2 (BPTT and BPTS). However we haven't covered the gradient descent algorithm and its different extensions. Gradient descent algorithm is an algorithm under active research. In this chapter it will be covered the process of updating the network parameters and the impact of selecting the right hyperparameters. This chapter will be divided into three sections, the first section will cover the different variants of gradient descent algorithm data selection 4.1, the second section will go through the most common gradient descent extensions4.2 and the third section will be about the neural network's hyperparameters4.2 which are vital to the neural network's adequate training.

4.1 Gradient Descent Variants

On this section, there will be covered three different variants of the gradient descent algorithms with regards to the amount of data they take to compute the gradient of the objective function. On the subsection 4.1.1 will be covered the Batch Gradient Descent which is the maximization of the likelihood over the entire training set, which is quite slow for big data sets. The second alternative is the Stochastic Gradient Descent (4.1.2) which depends on the error of one particular sample, this variant makes the computation much faster but it has to proceed through many iterations in order to show indications of improvement. The mini-batch Gradient Descent (4.1.3) is the compromise between the two variants discussed before. [69]

4.1.1 Batch Gradient Descent

The Batch Gradient Descent calculates the gradient of the objective function with regards to the parameters θ of the entire training set(4.1). Since each update is performed after having calculated the whole training dataset, it is easy to conclude that it is a quite slow way of training in cases that data sets are large. Moreover another disadvantage of this method is that it doesn't allow to update the model on an online way.

$$\theta = \theta - \eta * \nabla_\theta J(\theta) \tag{4.1}$$

4.1.2 Stochastic Gradient Descent

On the other hand, Stochastic Gradient Descent performs a parameter update for each sample i (4.2)[70]. This variant much faster that the Batch Gradient Descent and it is able to learn in an online way. However, this approach has a few flaws as well. During the first steps of the training, the objective function fluctuates heavily because the updates have high variance. This characteristic has strengths and weaknesses as well, because it can land to a potentially better local minima, or it can make the minimization process too complicated. Despite its high variance at the beginning of the training, on the long run it will become more stable.

$$\theta = \theta - \eta * \nabla_\theta J_i(\theta) \tag{4.2}$$

4.1.3 Mini-Batch Gradient Descent

Mini-Batch Gradient Descent[71] is the happy medium between the two contradictory variants discussed above(4.1.1,4.1.2). This approach performs the parameter updates based on the gradient of the parameter from n samples (batch size) of the training set (4.3). It tackles the problem of update's high variance which leads to a more stable convergence and it trains faster than the vanilla gradient descent.

$$\theta = \theta - \eta * \nabla_\theta J_{i:i+n}(\theta) \tag{4.3}$$

Despite its advantages, Mini-Batch Gradient Descent underlies some flaws. A key challenge of minimizing highly non-convex error functions com-

mon for neural networks is avoiding getting trapped in their numerous suboptimal local minima. Dauphin[72] argues that the difficulty arises in fact not from local minima but from saddle points, i.e. points where one dimension slopes up and another slopes down. These saddle points are usually surrounded by a plateau of the same error, which makes it notoriously hard for SGD to escape, as the gradient is close to zero in all dimensions.

4.2 Gradient Descent Extensions

There are several approaches for performing the parameter update. In this section it will be covered the most popular gradient descent optimization algorithms that are used to tackle the challenges that were mentioned at section 4.1. Those techniques are used to solve some of the problems that the vanilla gradient descent algorithm underlies. Namely, the challenge of finding a right learning rate (4.3.1), the learning rate is the same for each parameter which can be a problem if the data is sparse, being trapped to a suboptimal local minima (4.1.3). The techniques that will be discussed are focused on the learning rate manipulation and its impact on the parameters' update.

4.2.1 Vanilla update

Vanilla update is the simplest form of update, it performs the updates of the parameters towards the negative gradient direction. So far, when gradient descent algorithm was mentioned we were referring to the vanilla gradient descent(4.1,4.2,4.3).

4.2.2 Momentum update

Momentum update [73] speeds up when the parameter has a consistent gradient and slows down when the gradient changes directions. This approach was inspired by the physical perspective of the problem. Just like a ball rolling down a hill, the steepest the slope the faster the ball roles, based on the same idea the momentum update was conceived. The property of velocity is integrated by adding a fraction γ(momentum term) of the past update vector from the previous time step to the current update vector(4.4). After having calculated the update vector we integrate it into the parameter (4.5).

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \tag{4.4}$$

$$\theta = \theta - v_t \tag{4.5}$$

The outcome of using momentum is faster convergence and reduced oscillation[74]. At figure 4.1we can see the difference between the valilla gradient descent updates (on the left side) and the momentum updates (on the right side).

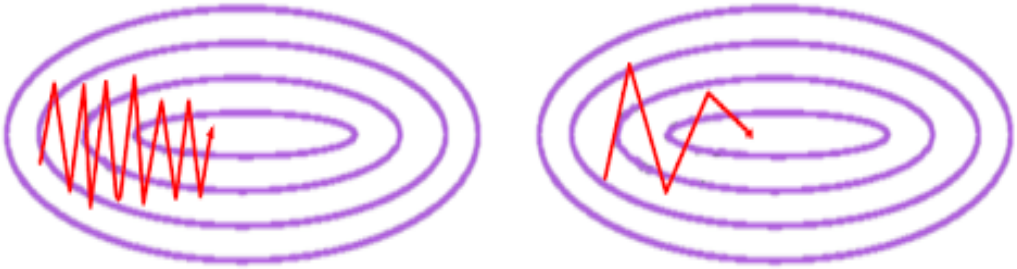


Figure 4.1: Vanilla vs Momentum

4.2.3 Nesterov Momentum update

Nesterov Momentum update is a smarter alteration of vanilla momentum update. The main idea of Nesterov Momentum is for the update to has a notion of where the gradient is heading therefore it slows down before the gradient changes direction. This technique is also known as Nesterov Accelerated Gradient (NAG) [75],[76] it makes a rough approximation of where the parameter will be, so now it can effectively look ahead by calculating the gradient with regards to the approximation of the future position and not the current one (4.6). According to Bengio[77] the estimated update prevents us from going too fast and results in increased responsiveness.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}) \tag{4.6}$$

$$\theta = \theta - v_t \tag{4.7}$$

The difference between the two approaches can be illustrated below at Figure 4.2.

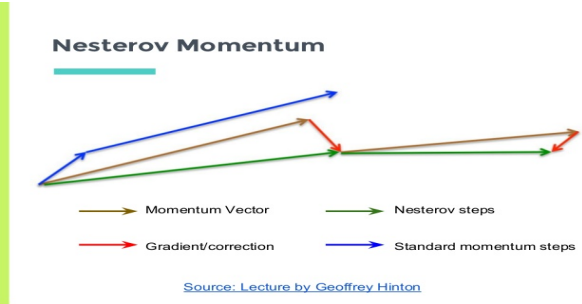


Figure 4.2: Momentum vs Nesterov Momentum

4.2.4 AdaGrad

AdaGrad[78] adapts the learning rate for every feature which eliminates the need of manually tuning the learning rate. It adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters. For this reason, it is well-suited for dealing with sparse data.[79] Moreover, Pennington [33] used Adagrad to train GloVe(section 2.3) word embeddings that we use at our experiment on the next chapter. The reason he used AdaGrad, is because infrequent words require much larger updates than frequent ones.

Since AdaGrad uses different learning rate for every parameter θ_i at every time step t for the sake of convenience we assume $g_{t,i}$ (4.8) to be the gradient of the objective function of parameter i and g_t(4.9) be the vector of all the parameters at time step t . Apart from the element-wise property of this approach, Duchi makes use of a diagonal matrix $G \in R^{dxd}$ that is consisted of the sum of the squares of the gradients with respect to the parameter up to time step t, while it also has a smoothing term ϵ that prevents it from being divided with 0. It is adapting the learning rate by caching the sum of squared gradients with respect to each parameter at each time step. The reason that we use the squared sums is not specified, but it performs way better than taking the matrix without the square root operation. Finally the formula for computing the parameter can be seen below.

$$g_{t,i} = \nabla_\theta J(\theta_i \tag{4.8}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{Gt + \epsilon}} \odot g_t \tag{4.9}$$

Despite its high performance, Adagrad has a serious drawback which stems from the fact that the learning rate shrinks after some point

due to its accumulation of the squared gradients in the denominator. This is the reason that it is considered to be notoriously aggressive according to the machine learning community.

4.2.5 AdaDelta

In order to tackle the learning rate shrinking problem Zeiler came up with a different way of adapting the learning rate. Adadelta [80] comes to rescue. Adadelta is an extension of Adagrad that alleviates its aggressively monotonically decreasing learning rate. It was suggested that instead of accumulating all the previous gradients it would be better to set a fixed window of size w and take the accumulated gradients of size w. Moreover, another alteration of Adagrad is the way it treats the past gradients. Instead of storing the past squared gradients, the sum of gradients is defined as a decaying average of all past squared gradients. The average of time step t depends only on the previous average and the current gradient. It is also used the momentum term ,that was covered at subsection 4.2.2, which determines how much off the past average will affect the current average.

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 \tag{4.10}$$

The parameter update is defined similar to the AdaGrad but instead of the diagonal matrix G_t we use the decaying average of the past squared gradients $E[g^2]_t$.

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{4.11}$$

Moreover the denominator of (4.11) is the root mean squared(RMS) error of criterion of the gradient, having noticed that Ziegel defined the decaying average of the squared parameter updates. However the $RMS[\Delta\theta]_t$ is not known so it is approximated with the RMS of the parameter updates from the previous time step. Therefore the updates of the parameters can be computed as :

$$\Delta\theta_t = \frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t \tag{4.12}$$

Something interesting about this approach is that the learning rate is irrelevant, as it is nowhere in the update rule.

4.2.6 RMSprop

RMSprop is an unpublished adaptive learning method from Hinton at his coursera Machine Learning course that is commonly used from the deep learning community (it is even a built-in function at tensorflow)tensorflow : python deep learning library, created by Google. RMSprop is very similar with AdaDelta, in fact is is just like the first part of AdaDelta but instead of γ there is a default value of 0.9 while its suggested initial learning rate is 0.001. Even though they were established the same period they were conceived independently.

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2 \tag{4.13}$$

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}}g_t \tag{4.14}$$

4.3 Hyperparameters

4.3.1 Learning Rate

Learning rate can be thought as the rate that the parameter update based on its gradient. Choosing a proper learning rate can be difficult. A learning rate that is too small leads to slow convergence, while a learning rate that is too large can make the loss function fluctuate around the minimum or even diverge.

4.3.2 Regularization

It is a method for preventing overfitting. It essentially works by setting a complexity penalty to the loss function. In practice, this means that it penalizes a function that is too non-linear and learns by heart the information that is contained in the training set and is not able to generalize well to new examples. In this paper we will mention the most popular regularizers

that we will also use at the experiments. Let those be L2 regularizer and dropout regularizer. But why do we really need regularization? It doesn't help the model perform well at the training set but perform well at the new examples (test set), which means that it minimizes the generalization error. The generalization error is the sum of the squared bias and variance of the trained model. The variance of the model indicates how much the model varies if we change the training set, the bias of the model is how close is the model to the true solution (the model that generated those instances).

L2 Regularization

The L2 regularization method [81] adds a regularization term in order to prevent the coefficients to fit so perfectly to overfit. When it comes to neural networks it only regularizes the connection weights the hidden layers. In more detail, what we do is to penalize the square of the weight value for each hidden layer k and for each connection i, j. Notice that the sum of i, j from equation (4.15) corresponds to the Frobenius Norm therefore it can be written as (4.16)

$$\Omega(\theta) = \sum_{k} \sum_{i} \sum_{j} (W_{i,j}^{(k)})^2 \tag{4.15}$$

$$\Omega(\theta) = \sum_{k} \|W^{(k)}\|_F \tag{4.16}$$

The gradient to the regularizer with respect to the k^{th} layer is two times the weight matrix :

$$\nabla_{w^{(k)}} \Omega(\theta) = 2W^{(k)} \tag{4.17}$$

It is important to notice that this is applied only at the weights, because we don't expect to overfit the training set by changing the biases a lot but more by changing the weights that really determine how the function can become more or less non-linear.

Dropout Regularization

In deep neural networks have been proposed two ways of dealing with overfitting , the first one is the unsupervised pre-training [82](which we will not discuss because it will not be used in this experiment therefore it is beyond

the scope of this paper), the second is dropout [83]. Dropout was proposed by Geoffrey Hinton as a technique for performing regularization. It works by randomly dropping nodes in a neural network and it emulates ensembles[1] of neural networks. The application can be seen at figures 4.3, 4.4 where Figure 4.3 illustrates a standard neural network while Figure 4.4 illustrates the very same neural network after having applied dropout. In practice, for each hidden unit once it have been computed we will independently set it to 0 (dropping out the value derived from the training) with a probability ,usually of 0.5. This process continues till we reach the output layer. Therefore as a result of the whole process the hidden units can't collaborate with each other in order to generate complex patterns that might be useful to fit the training data so they are forced to extract a feature that is useful in general.

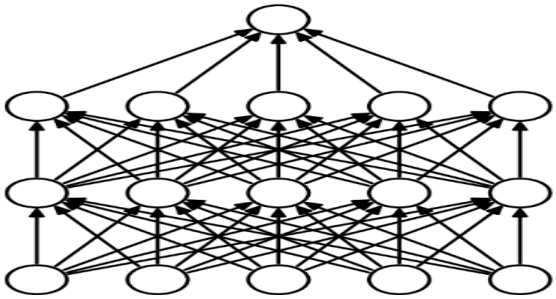


Figure 4.3: Standard Feed-forward Neural Network

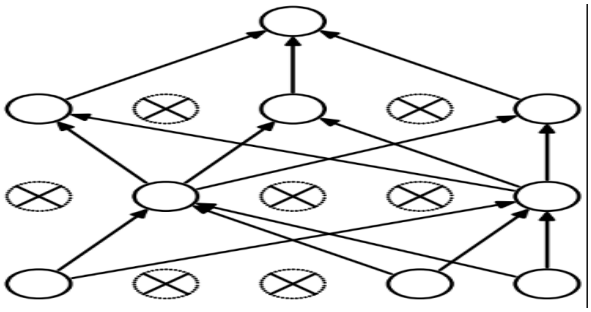


Figure 4.4: After applying Dropout

The dropout regularization technique, it has an impact on both the forward and backward propagation algorithm for training a neural network. In more detail, regarding the forward propagation, we set a random binary mask $m^{(k)}$ with values 0 (dropout the weight of the unit) or 1(retains the value), when it comes to the backward propagation, when we backpropagate the gradient till the preactivation($z1.3$) we also need to multiply it by the mask vector, due to the chain rule. This practically means that many gradients will be set to zero so the backpropagation won't flow through the hidden units that were dropped out.

[1] train a group of prediction models, then average their prediction or take the majority vote

Chapter 5

Experiments

This chapter is dedicated to the practical comparison between the Constituent LSTM, with the Tree-Based GRU (or in our case it is more appropriate to call it Constituent GRU). It is important to notice that the experiments have been conducted 5 times and the results are the product of the averaged results of all the trials. We use the Stanford Sentiment Treebank(SST), and we use the standard train/dev/test splits of 6920/872/1821 for the binary classification subtask and 8544/1101/ 2210 for the fine-grained classification subtask (there are fewer examples for the binary subtask since the neutral instances have been excluded). The sentiment label at each node is predicted using the classifier covered at the next subsection 5.1.1. Moreover the SST have each sentence structured as constituent parse trees, therefore we will use the Constituent LSTM as a comparison to our model. Please find the code necessary for running those experiments to the next url: `https://github.com/VasTsak/master_thesis`.

5.1 Model comparison

Before proceeding to the experiments we made the assumption that the Tree-based GRU will be faster to be trained because of its fewer parameters. The experiments proved us right. But training speed is just a factor (not even that critical) to select a model, what we really care about is its capability of being able to identify the underlying pattern just right, not learn the training set by heart (overfitting) nor ignoring some important features(underfitting).

5.1.1 Classification model

The goal of the paper is to compare the performance of the Tree-GRU architecture against the Tree-LSTM architecture on sentiment classification tasks. In practice, the model predicts a label $\hat{y}$ from a set of classes (2 for binary, 5 for fine grained) for some subset of nodes in a tree. The classifier and the objective function are exactly the same for both architectures. Let $\{x\}_j$ be the inputs observed at nodes in the subtree with root the node j.

$$\hat{p}_\theta(y \mid \{x\}_j) = softmax(W^p h_j + b^p), \tag{5.1}$$

$$\hat{y}_j = \underset{y}{\text{argmax}} \hat{p}_\theta(y \mid \{x\}_j). \tag{5.2}$$

Let m be the number of labeled nodes in the training set and the superscript k be the k^{th} labeled node, the cost function is:

$$J(\theta) = -\frac{1}{m} \sum_{k=1}^{m} \log \hat{p}_\theta(y^{(k)} \mid \{x\}^{(k)}) + \frac{\lambda}{2} \|\theta\|_2^2 \tag{5.3}$$

5.1.2 Binary Classification

The binary classification is a problem that classifies whether the sentiment of the sentence is positive or negative. The process of the training can be seen at the Figures 5.1, 5.2, and 5.3 where at Figure 5.1 it is plotted the average training time of each iteration and at Figures 5.2 are plotted the average loss of each iteration and at Figure 5.3 the training process[1] of Tree-GRU and Tree-LSTM.

All the plots have as their x-axis the number of epochs. The metrics that we plot are computed till the 12^{th} epoch and in cases of early stopping[2] we wouldn't take into account the 0 or "Non Assigned Number" of the trial that its training stopped earlier but we would just skip it and calculate the results based on the rest trials that had a full training process.

[1] training process: the training and validation scores at each epoch.

[2] early stopping: when the validation error increases for a specified number of iterations, the training process stops

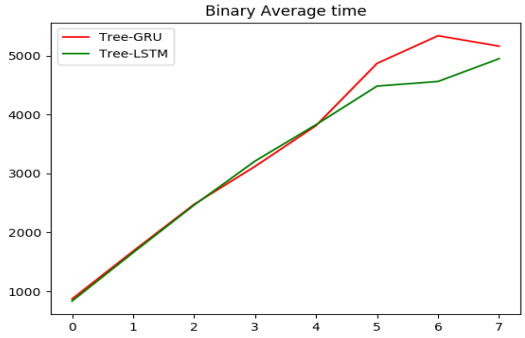


Figure 5.1: Binary Classification Average Training Time

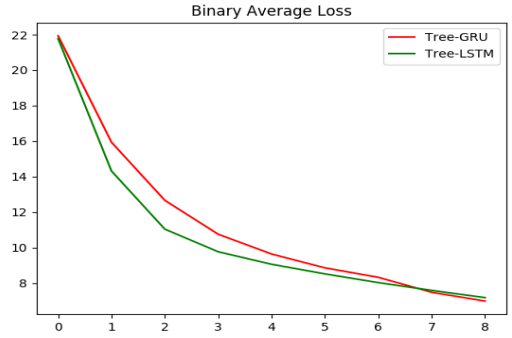


Figure 5.2: Binary Classification Average Training Loss

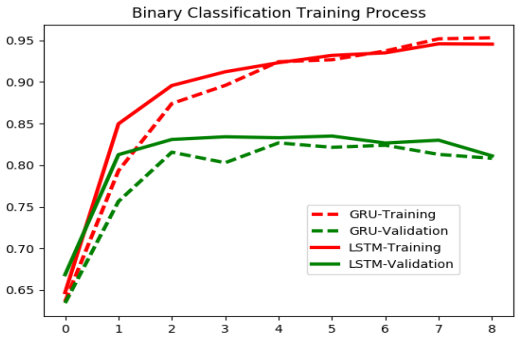


Figure 5.3: Binary Classification Training Process

5.1.3 Fine-grained Classification

The Fine-grained classification is a 5-class sentiment classification(1-Very Negative,2-Negative,3-Neutral,4-Positive,5-Very Positive). The experiment for the Fine-grained classification is under the same circumstances. The average time that each iteration lasted during the training process can be illustrated at Figure 5.4, the average loss that occured during the training process is illustrated at Figure 5.5 and the training process of the fine grained classification can be seen at Figure 5.6.

5.1.4 Hyperparameters and Training Details

We have initialized the word representations using the pre-trained 300-dimensional GloVe vectors[33].The training of the model was done with AdaGrad[78] and a learning rate of 0.05 also we used the mini-batch gradient

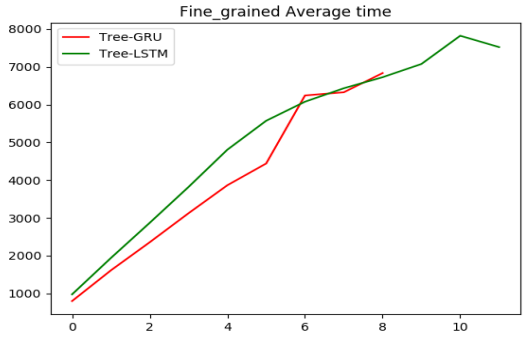


Figure 5.4: Fine Grained ClassificationAverage Training Time

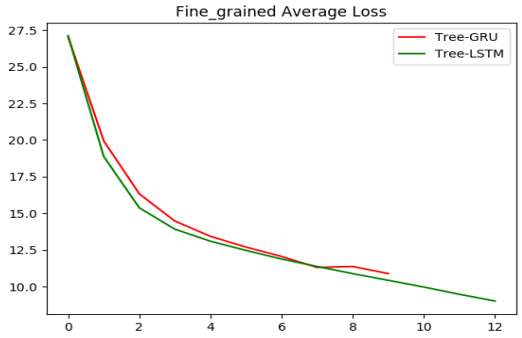
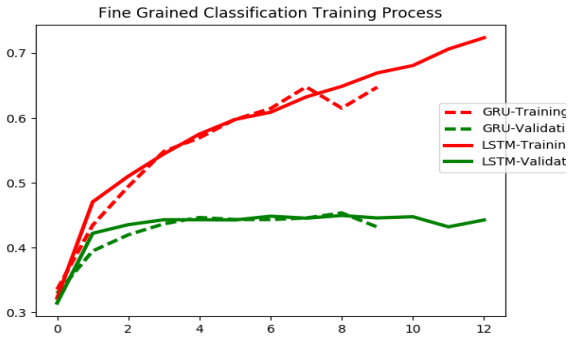


Figure 5.5: Fine Grained Classification Average Loss

Figure 5.6: Fine Grained Classification Training Process



descent algorithm with batch size of 25. The model parameters were regularized with L2 regularization strength of 0.0001 and dropout rate of 0.5. For the training process we have applied the early stopping technique in order to avoid overfitting. The goal of this paper is not to achieve a state-of-art accuracy but to make a critical comparison between the two models therefore we won't update the word representations during the training which boosts the accuracy approximately 0.05 (that is the accuracy boost gave to the Tree-LSTM).

5.2 Results

The results of both the binary and fine grained classification can be seen in Table 5.1 we can see that Tree-based GRU have slightly better performance with the tree-based LSTM, but it is important to notice from Table 5.2 the standard deviation of the individual predictions that the prediction

from Tree-GRU seem to be more fluctuate than the ones from Tree-LSTM therefore it is possible that this difference of performance can be random, because the standard deviation is quite high for the case of Tree-GRU.

Something important to notice about the training process of fine grained classification is that, the Tree-based GRU would stop at the 8^{th}iteration while the LSTM would go all the way till the 12^{th} iteration. Moreover something else to notice is that the Tree-LSTM for fine-grained classification seems like it has some more training to do before it overfits, in contrast with the Tree-GRU which would overfit before having executed twelve iterations, which can be observed above (Figures 5.6). This may have to do with the hyperparameters that we have chosen. We have set the early stopping at 2 iterations(as the Tree-based LSTM paper had), if we would set it to 3 the Tree-GRU may keep on training till the 12^{th} iteration.

MoreoverTree-GRU's training and validation scores seem to fluctuate more in the fine-grained classification 5.6 which may underlies unstable prediction and the need to train more.

Table 5.1: Sentiment Classification Accuracy

Model	Binary	Fine-grained
Tree-LSTM	84.43	45.71
Tree-GRU	85.61	46.43

Table 5.2: Sentiment Classification Standard Deviation

Model	Binary	Fine-grained
Tree-LSTM	0.35	0.55
Tree-GRU	0.93	0.98

5.3 Conclusions and Future Work

We can conclude that there is a difference in terms of performance ,not that significant though, between the tree-based LSTM and tree-based GRU. Moreover, Tree-based GRUs are trained faster -computationally- and they seem to converge faster so, the training process can stop earlier. Therefore it is a good alternative,if not a substitute. The area of Natural Language Processing is very active area of research, tree-based architectures proved to be very powerful for Natural Language Processing tasks, mostly because of their capability of handling negations. Many potential projects can be developed around Tree-Based GRUs, namely a Child-Sum approach,or the

of use unique reset and update gate for each child, or even try different GRU architectures [84]).

For the end, one philosophical thought. Can you imagine an entire system of neural networks performing different tasks, so that the end result is something actionable. Like building a brain, the language processing system would be just a small part, but you may have a neural network to do part of speech tagging another neural network to do name entity recognition and another network to parse sentences into trees. Even a more challenging problem might be to figure out what is the general architecture we can use so that we don't even have to tell the system to learn these things. In other words, a network of neural networks where each neural network can figure out what it should do on its own and be useful for the overall system in a global manner.

Appendix A

First Appendix

Let E is the error ,$\delta h_t = \frac{\theta E}{\theta h_t}$, we seek to find δo_t , δC_t, δi_t , $\delta \tilde{C}_t$, δf_t, , δC_{t-1}.

$$\delta o_t = \frac{\theta E}{\theta o_t} = \frac{\theta E}{\theta h_t}\frac{\theta h_t}{\theta o_t} = \delta h_t \odot tanh(C_t) \tag{A.1}$$

$$\delta C_t = \frac{\theta E}{\theta C_t} = \frac{\theta E}{\theta h_t}\frac{\theta h_t}{\theta C_t} = \delta h_t \odot o_t \odot (1 - tanh^2(C_t)) \tag{A.2}$$

$$\delta i_t = \frac{\theta E}{\theta i_t} = \frac{\theta E}{\theta C_t}\frac{\theta C_t}{\theta i_t} = \delta C_t \odot \tilde{C}_t \tag{A.3}$$

$$\delta \tilde{C}_t = \frac{\theta E}{\theta \tilde{C}_t} = \frac{\theta E}{\theta C_t}\frac{\theta C_t}{\theta \tilde{C}_t} = \delta C_t \odot i_t \tag{A.4}$$

$$\delta f_t = \frac{\theta E}{\theta f_t} = \frac{\theta E}{\theta C_t}\frac{\theta C_t}{\theta f_t} = \delta C_t \odot C_{t-1} \tag{A.5}$$

$$\delta C_{t-1} = \frac{\theta E}{\theta C_{t-1}} = \frac{\theta E}{\theta C_t}\frac{\theta C_t}{\theta C_{t-1}} = \delta C_t \odot f_t \tag{A.6}$$

Appendix B

Second Appendix

Given $\delta h_t = \frac{\theta E}{\theta h_t}$ we seek to find $\delta \tilde{h}_t h_t$, δr_t and δz_t

$$\delta \tilde{h}_t = \frac{\theta E}{\theta \tilde{h}_t} = \frac{\theta E}{\theta h_t} \frac{\theta h_t}{\theta \tilde{h}_t} = \delta h_t (1 - z) \tag{B.1}$$

$$\delta r_t = \frac{\theta E}{\theta r_t} = \frac{\theta E}{\theta \tilde{h}_t} \frac{\theta \tilde{h}_t}{\theta r_t} = \delta \tilde{h}_t \odot h_{t-1} U^h \odot (1 - \tilde{h}_t^2) \tag{B.2}$$

$$\delta z_t = \frac{\theta E}{\theta z_t} = \frac{\theta E}{\theta h_t} \frac{\theta h_t}{\theta z_t} = \delta h_t (h_{t-1} - \tilde{h}_t) \tag{B.3}$$

Bibliography

[1] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. *Acl (1)*, pages 1556–1566, 2015.

[2] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *arXiv*, pages 1–9, 2014.

[3] Bo Pang, Lillian Lee, and Shivakumar Vaithyanathan. Thumbs up?: sentiment classification using machine learning techniques. *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 79–86, 2002.

[4] Zachary Chase Lipton. A Critical Review of Recurrent Neural Networks for Sequence Learning. *CoRR*, abs/1506.0:1–38, 2015.

[5] Chomsky Noam. Syntactic Structures. 1958.

[6] Dan Klein and Christopher D Manning. A Generative Constituent-Context Model for Improved Grammar Induction. *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, (July):128–135, 2002.

[7] Muhua Zhu, Yue Zhang, Wenliang Chen, Min Zhang, and Jingbo Zhu. Fast and Accurate Shift-Reduce Constituent Parsing. *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 434–443, 2013.

[8] Michael a. Covington. A Fundamental Algorithm for Dependency Parsing. *Proceedings of the 39th Annual ACM Southeast Conference*, pages 95–102, 2001.

[9] Marie-Catherine De Marneffe, Bill MacCartney, and Christopher D. Manning. Generating typed dependency parses from phrase structure parses. *Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC 2006)*, pages 449–454, 2006.

[10] Dan Klein and Christopher Manning. Corpus-based induction of syntactic structure: models of dependency and constituency. *ACL '04: Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*, 1:478–485, 2004.

[11] Warren S Mcculloch and Walter Pitts. A logical calculus nervous activity. *Bulletin of Mathematical Biology*, 52(l):99–115, 1990.

[12] Frank. Rosenblatt. Principles of Neurodynamics. Perceptrons and the Theory of Brain Mechanisms. *Archives of General Psychiatry*, 7:218–219, 1962.

[13] David E. Rumelhart, Geoffrey E Hinton, and Ronald J. Williams. Learning representatons by back-propagating errors, 1986.

[14] H. Ritter and T. Kohonen. Self-organizing semantic maps. *Biological Cybernetics*, 61(4):241–254, 1989.

[15] J. J. Hopfield. Neural Networks and Physical Systems with Emergent Collective Computational Abilities, 1982.

[16] Paul J. Werbos. Backwards Differentiation in AD and Neural Nets: Past Links and New Opportunities. *Lecture Notes in Computational Science and Engineering*, 50:15–34, 2006.

[17] C M Bishop. Neural Networks for Pattern Recognition. 1995.

[18] David Zipser and Ronald J Williams. Gradient-Based Learning Algorithms for Recurrent Networks and Thei r Computational Complexity. *Back-propagation: Theory, Architectures and Applications*, pages 433–486, 1995.

[19] J.R. Firth. *Papers in linguistics, 1934-1951.* Oxford University Press, 1957.

[20] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, November 1975.

[21] Marco Baroni, Georgiana Dinu, and German Kruszewski. Don't count , predict ! A systematic comparison of context-counting vs . context-predicting semantic vectors. *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics.*, pages 238–247, 2014.

[22] Curt Burgess and Kevin Lund. The dynamics of meaning in memory. *In Cognitive Dynamics: Conceptual Change in Humans and Machines. Dietrich & Markman (Eds.), Psychology Press*, pages 117–156, 2000.

[23] San Diego, Linguistic Papers, and Hannah Rohde. Rhetorical questions as redundant interrogatives. *San Diego Linguistics Papers*, 2(2), 2006.

[24] Ronan Collobert. Rehabilitation of Count-based Models for Word Vector Representations. 2014.

[25] Scott Deerwester, Susan T Dumais, George W Furnas, and Thomas K Landauer. Indexing by Latent Semantic Analysis. *Society*, 41(6):391–407, 1990.

[26] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural Language Processing (Almost) from Scratch. *Journal of Machine Learning Research*, 12:2493–2537, 2011.

[27] Andriy Mnih. Learning word embeddings efficiently with noise-contrastive estimation. *Nips*, pages 1–9, 2013.

[28] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[29] Omer Levy, Yoav Goldberg, and Ido Dagan. Improving Distributional Similarity with Lessons Learned from Word Embeddings. *Transactions of the Association for Computational Linguistics*, 3:211–225, 2015.

[30] G. H. Golub and C. Reinsch. Singular value decomposition and least squares solutions. *Numerische Mathematik*, 14(5):403–420, 1970.

[31] G H Golub and C F Van Loan. Matrix Computations, 1996.

[32] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed Representations of Words and Phrases and their Compositionality. *Nips*, pages 1–9, 2013.

[33] Jeffrey Pennington, Richard Socher, and Christopher D Manning. GloVe : Global Vectors for Word Representation. pages 1532–1543, 2014.

[34] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[35] MI I Jordan. Serial order: A parallel distributed processing approach, 1986.

[36] Kevin J. Lang. *A Time Delay Neural Network Architecture for Speech Recognition*. PhD thesis, Pittsburgh, PA, USA, 1989. AAI9011852.

[37] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[38] Herbert Jaeger. The echo state approach to analysing and training recurrent neural networks with an Erratum note 1. *GMD Report*, (148):1–47, 2010.

[39] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Gated feedback recurrent neural networks. *Proceedings of the 32nd International Conference on Machine Learning, {ICML} 2015*, 37:2067—-2075, 2015.

[40] Jane Elith, Catherine H. Graham, Robert P. Anderson, Miroslav Dudík, Simon Ferrier, Antoine Guisan, Robert J. Hijmans, Falk Huettmann, John R. Leathwick, Anthony Lehmann, Jin Li, Lucia G. Lohmann, Bette A. Loiselle, Glenn Manion, Craig Moritz, Miguel Nakamura, Yoshinori Nakazawa, Jacob McC. M. Overton, A. Townsend Peterson, Steven J. Phillips, Karen Richardson, Ricardo Scachetti-Pereira, Robert E. Schapire, Jorge Soberón, Stephen Williams, Mary S. Wisz, and Niklaus E. Zimmermann. Novel methods improve prediction of species' distributions from occurrence data. *Ecography*, 29(2):129–151, 2006.

[41] Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-efficient backpropagation through time. *CoRR*, abs/1606.03401, 2016.

[42] Sepp Hochreiter. Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies. 2001.

[43] Y Bengio. Learning long-term dependencies with gradient descent is difficult, 1994.

[44] Juergen Jürgen Schmidhuber. Learning Complex, Extended Sequences Using the Principle of History Compression. *Neural Comput.*, 4(2):234–242, 1992.

[45] James Martens. Deep learning via Hessian-free optimization. *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 951:735–742, 2010.

[46] Herbert Jaeger, Wolfgang Maass, and Jose Principe. Special issue on echo state networks and liquid state machines, 2007.

[47] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

[48] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.

[49] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech Recognition With Deep Recurrent Neural Networks. *Icassp*, (3):6645–6649, 2013.

[50] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. *CoRR*, abs/1411.4555, 2014.

[51] Wojciech Zaremba and Ilya Sutskever. Learning to execute. *CoRR*, abs/1410.4615, 2014.

[52] F.A. Gers and J. Schmidhuber. Recurrent nets that time and count. *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, 1:189–194 vol.3, 2000.

[53] Felix a Gers, Nicol N Schraudolph, and Jurgen Schmidhuber. Learning Precise Timing with LSTM Recurrent Networks. *Journal of Machine Learning Research*, 3(1):115–143, 2002.

[54] F. Robinson, A. J. Fallside. The utility driven dynamic error propagation network. 1987.

[55] Richard Socher, Eh Huang, and Jeffrey Pennington. Dynamic Pooling and Unfolding Recursive Autoencoders for Paraphrase Detection. *Advances in Neural Information Processing Systems*, pages 801–809, 2011.

[56] J. B. Pollack. Recursive distributed representations. *Artif. Intell.*, 46(1-2):77–105, November 1990.

[57] C. Goller and A. Kuchler. Learning task-dependent distributed representations by backpropagation through structure. In *Neural Networks, 1996., IEEE International Conference on*, volume 1, pages 347–352 vol.1, Jun 1996.

[58] F Costa, Paolo Frasconi, V Lombardo, and G Soda. Towards incremental parsing of natural language using recursive neural networks. *Applied Intelligence*, 19(1-2):9–25, 2003.

[59] G Hinton. Mapping Part-Whole Hierachies into Connectionist Networks. *Connectionist Symbol Processing*, 46(1990):47–75, 1990.

[60] Richard Socher, Alex Perelygin, and Jy Wu. Recursive deep models for semantic compositionality over a sentiment treebank. *Proceedings of the ...*, pages 1631–1642, 2013.

[61] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A Neural Probabilistic Language Model. *The Journal of Machine Learning Research*, 3:1137–1155, 2003.

[62] James Henderson. Neural network probability estimation for broad coverage parsing. *Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics - EACL '03*, 1(March):131–138, 2003.

[63] Ronan Collobert and Jason Weston. A unified architecture for natural language processing. *Proceedings of the 25th international conference on Machine learning - ICML '08*, 20(1):160–167, 2008.

[64] Richard Socher, Christopher D Cd Manning, and Andrew Y Ay Ng. Learning continuous phrase representations and syntactic parsing with recursive neural networks. *Proceedings of the NIPS-2010 Deep Learning and Unsupervised Feature Learning Workshop*, pages 1–9, 2010.

[65] Ben Taskar, Dan Klein, Michael Collins, Daphne Koller, and Christopher Manning. Max-Margin Parsing. *Proc. EMNLP*, pages 1–8, 2004.

[66] Richard Socher, Brody Huval, Christopher D Manning, and Andrew Y Ng. Semantic Compositionality through Recursive Matrix-Vector Spaces. *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, (Mv):1201–1211, 2012.

[67] Richard Socher. Recursive Deep Learning for Natural Language Processing and Computer Vision. *PhD thesis*, (August), 2014.

[68] Lon Bottou. *Stochastic Gradient Tricks*, volume 7700, page 430445. Springer, January 2012.

[69] Andrew Ng. 1. Supervised learning. *Machine Learning*, pages 1–30, 2012.

[70] León Bottou. Large-Scale Machine Learning with Stochastic Gradient Descent. *Proceedings of COMPSTAT'2010*, pages 177–186, 2010.

[71] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. Efficient Mini-batch Training for Stochastic Optimization. *Kdd*, pages 661–670, 2014.

[72] Yann Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *arXiv*, pages 1–14, 2014.

[73] Richard S Sutton. Learning to Predict by the Method of Temporal Differences. *Machine Learning*, 3(1):9–44, 1988.

[74] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1):145 – 151, 1999.

[75] Yurii Nesterov. A method of solving a convex programming problem with convergence rate o (1/k2). In *Soviet Mathematics Doklady*, volume 27, pages 372–376, 1983.

[76] Ilya Sutskever. Training Recurrent neural Networks. *PhD thesis*, page 101, 2013.

[77] Yoshua Bengio, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. Advances in optimizing recurrent networks. *CoRR*, abs/1212.0901, 2012.

[78] John Duchi. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization . 12:2121–2159, 2011.

[79] A. Coates, B. Carpenter, C. Case, S. Satheesh, B. Suresh, T. Wang, D. J. Wu, and A. Y. Ng. Text detection and character recognition in scene images with unsupervised feature learning. In *2011 International Conference on Document Analysis and Recognition*, pages 440–445, Sept 2011.

[80] Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.

[81] Arthur E Hoerl and Robert W Kennard. Ridge Regression: Application to nonorthogonal problems. *Technometrics*, 12(1):69–82, 1970.

[82] Dumitru Erhan, Aaron Courville, and Pascal Vincent. Why Does Unsupervised Pre-training Help Deep Learning ? *Journal of Machine Learning Research*, 11:625–660, 2010.

[83] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012.

[84] Alexey Dosovitskiy, Jost Tobias Springenberg, and Thomas Brox. Learning to generate chairs with convolutional neural networks. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 07-12-June:1538–1546, 2015.

[85] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv:1502.03167*, pages 1–11, 2015.

[86] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout : A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research (JMLR)*, 15:1929–1958, 2014.

[87] Sepandar D. Kamvar, Dan Klein, and Christopher D. Manning. Spectral learning. *IJCAI International Joint Conference on Artificial Intelligence*, pages 561–566, 2003.

[88] Melvin Johnson, Mike Schuster, Quoc V Le, Maxim Krikun, Yonghui Wu, Zhifeng Chen, Nikhil Thorat, Fernanda Viégas, Martin Wattenberg, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's Multilingual Neural Machine Translation System: Enabling Zero-Shot Translation. pages 1–16.

[89] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *CoRR*, abs/1502.0, 2015.

[90] Alex Graves. Generating Sequences with Recurrent Neural Networks. *Technical Reports*, pages 1–43, 2013.

[91] Alex Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*. 2008.

[92] Razvan Pascanu, Çaglar Gülçehre, Kyunghyun Cho, Yoshua Bengio, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. How to Construct Deep Recurrent Neural Networks. *CoRR*, abs/1312.6:1–10, 2013.

[93] Scott Reed and Nando de Freitas. Neural Programmer-Interpreters. pages 1–13, 2015.

[94] Hazuki Tachibana, Wanxiang Che, Freeman Zhang, Deep Learning, Daniel Andrade, , , Richard Socher, Deep Learning, and Natural Language Processing. Recursive Neural Networks 236. , pages 1–92, 2016.

[95] Alex Graves, Navdeep Jaitly, and Abdel Rahman Mohamed. Hybrid speech recognition with Deep Bidirectional LSTM. *2013 IEEE Workshop on Automatic Speech Recognition and Understanding, ASRU 2013 - Proceedings*, pages 273–278, 2013.

[96] Ian Goodfellow. NIPS 2016 Tutorial: Generative Adversarial Networks. 2016.

[97] Jonas Gehring, Michael Auli, David Grangier, and Yann N Dauphin. a Convolutional Encoder Model for Neural Machine Translation. pages 1–13, 2017.

[98] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *International Conference on Learning Representations (ICRL)*, pages 1–14, 2015.

[99] Bram Bakker. Reinforcement Learning with Long Short-Term Memory. *Advances in Neural Information Processing Systems 14*, pages 1475–1482, 2002.

[100] Frbmi Jordan. Learning spectral clustering. *Advances in Neural Information Processing Systems 16 (NIPS)*, pages 305–312, 2004.

[101] Anton Maximilian Schäfer. Reinforcement Learning with Recurrent Neural Networks. *Thesis*, pages 410–420, 2008.

[102] D. Prescher, R. Scha, K. Simaan, and A. Zollmann. What are treebank grammars? *Belgian/Netherlands Artificial Intelligence Conference*, 2006.

[103] Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling Network Architectures for Deep Reinforcement Learning. *arXiv*, (9):1–16, 2016.

[104] David Money Harris and Sarah L Harris. *Digital design and computer architecture*. 2012.

[105] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks.

[106] G Salton, a Wong, and C S Yang. A Vector Space Model for Automatic Indexing. *Magazine Communications of the ACM*, 18(11):613–620, 1975.

[107] Ozan Irsoy and Claire Cardie. Deep recursive neural networks for compositionality in language. *Advances in Neural Information Processing Systems*, 3(January):2096–2104, 2014.

[108] F Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386–408, 1958.

[109] Wei Liu, Jun Wang, and Shih Fu Chang. Robust and scalable graph-based semisupervised learning. *Proceedings of the IEEE*, 100(9):2624–2638, 2012.

[110] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional LSTM networks. *Proceedings of the International Joint Conference on Neural Networks*, 4:2047–2052, 2005.

[111] Thomas Voegtlin and Peter F. Dominey. Linear recursive distributed representations. *Neural Networks*, 18(7):878–895, 2005.