Institutionen för datavetenskap
Department of Computer and Information Science

Examensarbete

Neural Network for Dependency Parsing

by

Zonghan Wu

LIU-IDA/LITH-EX---16/001--SE

2016-06-17

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Linköpings universitet
Institutionen för datavetenskap

Examensarbete

Neural Network for Dependency Parsing

by

Zonghan Wu

LIU-IDA/LITH-EX---16/001--SE

2016-06-17

Handledare: Marco Kuhlmann
Examinator: Oleg Sysoev

Abstract

Neural network with the assist of word embeddings alleviates the labor of feature engineering in the field of dependency parsing. Millions of artificially designed features can be reduced to a number within a thousand. Though a neural network saves the dimension of data inputs dramatically, the number of parameters to be estimated is still considerable. This thesis implemented a graph-based dependency parser with a simple feedforward neural network. Experimental results show that our neural network is effective in capturing syntactic structures of sentences. The graph-based dependency parsing is generalized as a structured prediction problem under the framework of statistical learning. We introduced the feedforward neural network as a generative model with the aid of basis functions. This neural network is also applicable to common classification problems. Exact inference is intractable because of the complexity of output space, the number of parameters and the size of data. We trained our neural network by minimizing the upper bound of posterior expected loss on a training set which is equivalent to a so called max-margin principle. The margin loss function in the training objective is interpreted as the empirical loss. The model is expected to perform differently with a different margin loss function. We experimented with variations of margin loss functions however did not find any significant difference in terms of model accuracy.

Acknowledgments

I am grateful for my supervisor Marco Kulhmann's great guidance on my thesis. You taught me to do things with a easy start and a hard end though this thesis is not called an end. I would like to thank professor Mattias Villani for pushing me think neural network in a statistical way. It changed the view how I see problems in my thesis. I would like to thank my colleague Robin for his inspiration and his advice. You are always willing to help me. I would like to thank my opponent Hector. You gave me as much feedback as you can. I would like to thank Malte for his support and encouragement. We walked the journey together. Last but not least I would like to thank all the teachers and friends who I met in Sweden. You are my best memory.

Contents

List of Figures

List of Tables

1 Introduction

Structured prediction, less known than regression and classification, is a branch of supervised learning that builds a mapping from input space to output space. Rather than predicting a real value, either continuous or discrete, structured prediction predicts structured data types such as sequences and graphs. In natural language processing, structured prediction is of particular importance. Many tasks involves structured prediction such as speech recognition, part-of-speech tagging and dependency parsing. Traditionally, a structured prediction model trains a function which scores how likely a possible structure y is for x. In prediction, the problem becomes a combinatorial optimization task. It searches for a structure with the highest score among all the other candidates in the finite output space. As shown in Equation 1.1,

$$\hat{y} = \underset{y \in Y(x)}{\operatorname{argmax}}\, Score(x, y; \theta) \tag{1.1}$$

where $Y(x)$ is the output space given x and θ represents model parameters. In this thesis, we explore the use of structured prediction in the context of natural language processing. In dependency parsing, a structured prediction model maps a sentence x to a representation of its syntactic structure in the form of a dependency graph which is a directed graph, as shown in Figure 1.1. In an arc, the word which the arc points from is called the head and the word which the arc points to is called the dependent. For example, in Figure 1.1, the head of "Many" is "investors". The output space of a sentence consists of all possible directed graphs satisfying certain constraints (these constraints will be explained in Section 2.1). By putting these constraints, a combinatorial optimization algorithm called Eisner algorithm guarantees to find a highest scoring graph with scalable time.

Approaches taking dependency parsing as a structured prediction problem are called graph-based approaches. Conventional graph-based approaches in dependency parsing commonly adopt a linear model as the score function [1] [2] [3] [4]. Features are extracted through representing words by boolean variables. Millions of handcrafted features are created as the number of features is proportional to the size of a vocabulary. On the one hand, the design of these handcrafted features heavily rely on expert linguistic knowledge. On the other hand, a linear model lacks of generality when a decision boundary is not linear separable. Very recently, Pei et al. [5] first proposed a neural network model as the score function for graph-based dependency parsing. The use of neural networks greatly released the labor of feature engineering as neural networks are able to learn low dimension word representations.

It to a large extent alleviates researchers' burden in exploiting good features and enables researchers focus more on model architecture design. With the assist of neural networks, it is promising to solve dependency parsing problems with a more complex search space. However, Pei et al.'s model is over complicated in a way that they trained two neural networks with a same architecture to score arcs of different directions. Arc directions can be seen in Figure 1.1. For example. "many←investor" is a left arc and "believe→is" is a right arc. In Pei et al.'s model, a left arc would be scored differently than an right arc even though nodes of two arcs are identical. Following Pei et al., Kiperwasser et al. [6] introduced a simpler neural network architecture which only takes features related with a head node and a dependent node as inputs without considering arc directions. Kiperwasser et al.'s model achieves a good performance, showing the possibility that arc directions are not important features. However the inputs in Kiperwasser et al.'s neural network model are outputs of an extra LSTM model which is inherently complicated. A research question comes naturally to ask how the model performance will be affected if we use features chosen by Pei et al. but only use one neural network instead of two. This simplification is worth to investigate since it will reduce about 200,000 parameters in the model.

Both Pei et al. [5] and Kiperwasser et al. [6] presented their models from a non-probabilistic view. The training objective of their models is the max margin principle. The idea is that the score of a correct tree should be at least larger than the score of an wrongly predicted tree plus a margin loss (see Equation3.7). The margin loss is only incurred if both the head of a word and their dependency type are incorrectly predicted. However, under the framework of statistical learning, the margin loss is interpreted as the empirical loss. This allows us to vary the margin loss according to actual situations. In dependency parsing, we would expect that the empirical loss should be lower if the head of a word is correctly predicted with an incorrect dependency type than the loss incurred if both the head and the dependency type are predicted wrong. A second research question is how variations of margin loss will affect the model performance.

In this thesis, we explore the possibility of simplifying the neural network model proposed by Pei et al. [5] and use the Eisner Algorithm as the combinatorial optimization method. Main contributions of this thesis are

- Reduced the model complexity of Pei et al.'s work [5] with an acceptable model accuracy.
- First introduced the feedforward neural network model as a generative model in the context of statistical learning(in Section 3.3).
- Proposed the Loss-Augmented Eisner Algorithm(in Section 3.2).
- Implemented a graph-based dependency parsing system.

This thesis is organized as follows. In Chapter 2, a framework of a dependency parsing system is presented. Main components of the dependency parser are introduced. In Chapter 3, we introduce methods of training a score function and present the work flow of our dependency parser. In Chapter 4, we describe training data, report experimental results, and make a comparison with Pei et al.'s model. In Chapter 5 potential problems and future work are discussed. In Chapter 6 conclusions in this thesis are summarized.

1.1 Background

Analyzing a sentence's syntactic structure is a first step towards understanding the meaning of a sentence. Syntactic structure is the set of all words connections in a sentence. Words are the smallest elements in linguistics which express meaning. A sentence which wraps a sequence of words can deliver broader meanings as words ceases to be isolated and creates

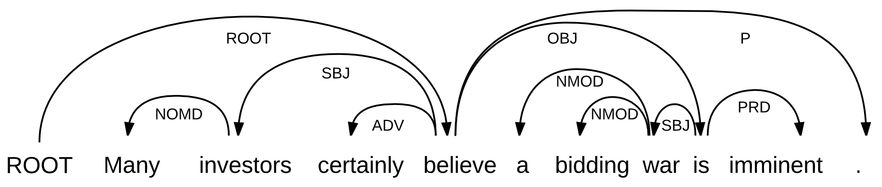

ROOT
SBJ
NOMD
ADV
OBJ
P
NMOD
NMOD
SBJ
PRD
ROOT Many investors certainly believe a bidding war is imminent .

Figure 1.1: An example of a dependency graph

connections with its neighboring words. Words can be considered as functional objects which internally define certain rules and distributions of connecting with other words. The deterministic rules are a subset of grammar knowledge. For example, “the” can only modify a noun. The probabilistic distributions are reflections of words’ semantic information. For example, “bread” can hardly be the subject of “eat” unless it is assumed to be a kind of living although it is grammatically correct.

In dependency parsing, the syntactic structure of a sentence is represented as a dependency tree, a directed graph over the words in a sentence. A directed arc points from the head node to the dependent node. The label of the arc defines the dependency type between the head and the dependent. For example, in Figure 1.1, “many” functions as the noun modifier of “investors”,“certainly” functions as the adverb modifier of “believe” and “investors” functions as the subject of “believe”.

Dependency Parsing Dependency parsing is a fundamental task in NLP which automatically predicts a sentence’s syntactic structure. It is widely integrated into other NLP systems such as sentence compression, information retrieval, and machine translation. Sentence compression is a crucial procedure for automatically summarizing articles. The core constitutes of a sentence consists of subject, object, and predicate arguments. Sentence compression retains the core constitutes of a sentence and removes words that modify core constitutes. For example, the sentence “I drink green tea in the sunshine” is abbreviated as “I drink tea”. Applying dependency parsing is a direct way to find core constitutes in a sentence. In information retrieval, dependency parsing opens the possibility to query information connected with a certain word. For example, one can possibly query the object of “buy” in companies’ annual reports in order to know what properties have been acquired during this year. This might help investors quickly locate valuable information in thousands of annual reports. In machine translation, syntactic structures in source language and in target language sometimes are different. With the assist of dependency parsing, a system could naively translate a sentence word by word and transform the syntactic structure of the source language to the target language. For example, in English the sentence “I go first” will be literally translated into Mandarin as “I first go”.

There are two main approaches in dependency parsing. One is called the transition-based approach. Another is called the graph-based approach. Transition-based approach builds a transition system which makes a prediction locally. It starts from an initial state, predicts the next transition given input and the parsing history until the terminal state is reached. Pioneer researches in transition-based approaches are Nivre [7] and Nivre [8]. The graph-based approach generalizes dependency parsing as a structured prediction problem. It learns a score function which measures the goodness of a candidate tree and parse a sentence by a graph search algorithm given the score function. Depending on different assumptions about a search space, a dependency tree can be projective or non-projective, which will be explained in Section 2.1. Graph search algorithms mainly include the CKY algorithm [9], the Eisner algorithm [10] for projective trees and C-L-E algorithm [11] for non-projective trees. Learning methods falls into three categories, perceptron based, max-margin based and probabilistic based methods.

The averaged perceptron algorithm proposed by Collin and Roark [12] memorizes the features occurred in a correct tree and penalizes the features occurred in a wrong tree. Taskar et al. [13] introduced max margin principle inspired by support vector machine to the parsing problem. It states that the score of a correct tree should be larger than the score of an incorrect tree up to a margin. McDonald et al. [1] combined a max-margin based algorithm (MIRA) with Eisner algorithm to parse projective trees. McDonald et al. [2] generalized the parsing problem as searching for the maximum spanning tree. They replaced the Eisner algorithm in the paper [1] with the C-L-E algorithm to parse non-projective trees. McDonald et al.'s work [2] assumed the score of a tree is factored into scores of individual arcs. However the score of a tree can also be factored into scores of higher order subgraphs. Mcdonald and Pereira [3], Koo and Collins [4] tackled the issue of second order factorization and third order factorization respectively. Probabilistic methods in dependency parsing treat the score function as a joint or conditional probability. Most probabilistic methods assumed the log-linear generative model. The generative approach models the joint probability of a tree and a sentence, and then derive the conditional probability of the tree given the sentence. McDonald and Satta [14], Koo et al. [15], Smith and Smith [16] applied the matrix tree theorem by Tutte [17] to derive the summation over probabilities of all possible trees assuming non-projectivity. For projective trees, Paskin [18] explored the inside-outside algorithm by Lari and Young [19].

Neural Networks Neural networks has experienced ups and downs in the history. In 1943, Warren McCulloch and Walter Pitts's paper [20] represents the start of researches about neural network models. In 1974, Werbos [21] invented the back-propagation algorithm to train neural network in his doctor dissertation. Rumelhart and et al. [22] rediscovered back-propagation algorithm and made it well known to the academia. Feedforward neural networks with additional hidden layers empirically does not gain extra benefits. This was confirmed by Hornik and et al. [23]. They proved that multilayer feedforward neural networks with a single hidden layer of enough hidden units are universal approximators of any square integral functions. In the 90s, neural networks in machine learning received less interest due to the limitation of computer speed and dataset volume. Entered into the new millennium, powered by the development of computer technology such as multiprocessor graphics cards, neural networks in a real sense started to shine. Several methods such as AdaGrad [24], dropout [25], and batch normalization [26] proposed in the last five years also greatly reduced the generalization error of neural networks.

In natural language processing, neural networks are capable of learning word representations. Bengio et al. [27] introduced a neural probabilistic model to learn vector representations of words in a low dimension. Words in a similar context would result in a closer vector representations. The low-dimension word representation known as word embeddings preserved abstract syntactic and semantic information learned through a large text corpora. Prominent works are followed by Mikolov et al. [28] and Pennington et al. [29]. They introduced word2vec word embeddings and GloVe word embeddings respectively. Collobert et al. [30] presented a unified neural network framework for solving natural language processing tasks with the aid of word embeddings. It popularizes the use of neural networks in NLP research. Fostered by the power of neural networks, the ability to generalize non-linearity and the ability to learn low-dimension word representations, researchers very recently applied neural networks to dependency parsing, Chen et al. [31] in a transition-based approach, Pei et al. [5] in a graph-based approach and Kiperwasser et al. [6] in both approaches.

1.2 Notation

This section defines the basic notation used throughout this thesis. The input of dependency parsing is a sentence x. Let x_i denote the i^{th} word in the sentence. The desired output y is a sequence of tuples (y_{i1}, y_{i2}) where y_{i1} is the head word of x_i and y_{i2} is the index of dependency type between the arc (x_i, y_{i1}). The index of dependency type is given by alphabetical order. For example, in Figure 1.1, "certainly" is the 4th word in the sentence. Its head is "believe". The index of their dependency type "ADV" is 1. Accordingly, $x_4 =$"certainly",$y_{41} =$"believe", and $y_{42} = 1$.

2 Framework

In Introduction, we mentioned that given a trained score function, dependency parsing is a combinatorial optimization task. A dependency parsing system searches the dependency tree with the highest score as a prediction for a sentence. As shown in Equation 2.1,

$$\hat{y} = \underset{y \in Y(x)}{\operatorname{argmax}}\, Score(x, y; \theta) \tag{2.1}$$

where $Y(x)$ is the search space given x and θ represents model parameters.

In this Chapter, we present the framework of a dependency parser. Equation 2.1 indicates that a dependency parser consists of three main components, a defined search space Y(x), a graph search algorithm and an actual form of the score function. Section 2.1 defines the search space of a sentence syntactic structure in this thesis. Section 2.2 describes the Eisner algorithm. The Eisner algorithm decomposes the score of a dependency tree into scores of individual arcs. The score of a dependency tree is the sum of scores of individual arcs. The Eisner algorithm is the graph search algorithm which searches the highest scoring tree within the search space defined in Section 2.1. In this thesis, the form of the score function is a feed-forward neural network. Section 2.3 gives some background knowledge about feedforward neural networks. Methods about how to train the score function will be introduced in Chapter 3. With a defined search space, a graph search algorithm, and a trained score function, the work flow of a dependency parser in prediction is drafted in Figure 2.1. For readers who need a fast reading, the following sections in Chapter 2 can be skipped.

2.1 Dependency Tree

The search space of our dependency parser is the set of all possible projective dependency trees. We summarized relevant knowledge about dependency trees from Kübler et al. [32].

2.1.1 Sentence

A sentence is an ordered set of tokens, denoted by $S = x_0, x_1, ..., x_n$

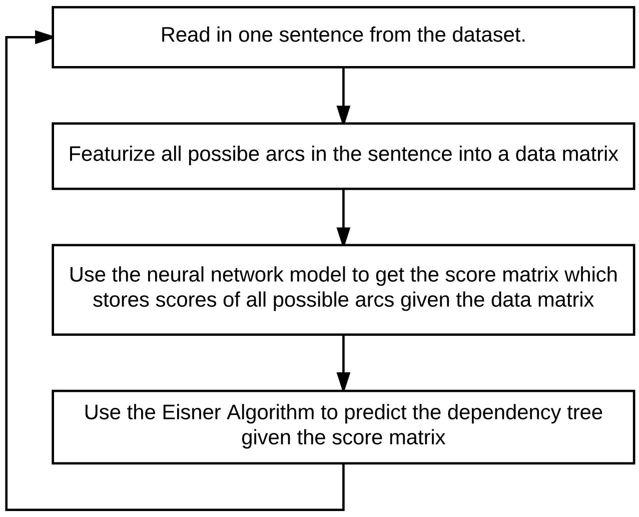

Read in one sentence from the dataset.
Featurize all possibe arcs in the sentence into a data matrix
Use the neural network model to get the score matrix which stores scores of all possible arcs given the data matrix
Use the Eisner Algorithm to predict the dependency tree given the score matrix

Figure 2.1: Work Flow of a Dependency Parser in Prediction

2.1.2 Dependency Relation

A dependency relation is a triple of (x_i, x_j, r). The triple represents a labeled directed arc pointing from head x_i to dependent x_j with relation type r,

$$x_i \xrightarrow{r} x_j$$

where $r \in R$, R is the set of all dependency types and $(x_i, x_j, r) \in A$, A is the set of all dependency relations in a sentence. To give an example, in Figure1.1 ("investors","many","NMOD") forms a dependency relation. It points from "investors" to "many" with dependency type "NMOD".

2.1.3 Dependency Tree

A dependency tree is a labeled directed acyclic graph $G = (S, A)$ originating from the root x_0. It satisfies the following four properties:

1. Root Property
   There does not exist a head for the root node of a dependency tree. The root node of a sentence is the word which does not modify any other words and is only modified by other words. For computational convenience, it is practical to insert an artificial node $x_0 =< ROOT >$ which points to the real root word of the sentence with relation type "ROOT".

2. Connectedness Property
   There does not exist an isolated node. That is, regardless the direction of arcs, there is a path between every pair of node in the graph. This property assumes that every word in a sentence always have some relations with other words.

3. Single-head Property
   All nodes have and only have one head except x_0. Since x_0 is an artificial node, it implies all words of a sentence have and only have one head.

4. Acyclicity property
   There does not exist any circles. A circle would imply that a word is dependent on itself which is meaningless.

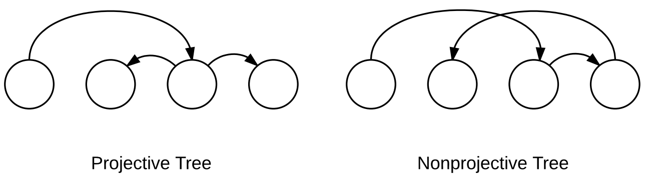


Figure 2.2: Demonstrations of Projective Trees and Non-projective Trees

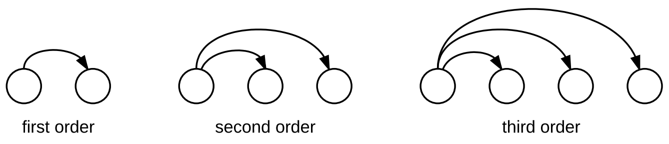


Figure 2.3: Subgraph Orders

2.1.4 Projectivity and Non-projectivity

Dependency trees fall into two categories, projective trees and non-projective trees. Projective trees satisfy the property that for all arcs in the projective tree there is a path from the head of the arc to any nodes in between the head and the dependent. Intuitively it means that the projective tree does not contain any crossing arcs and vice versa for non-projective trees. The phenomenon of non-projectivity seldom occurs in English while in Czech, Dutch and German it occurs more frequently. Figure 2.2 gives examples of a projective tree and a non-projective tree. As this thesis mainly focus on English sentences, we assumed the search space for a sentence is the set of all possible projective dependency trees.

2.2 Eisner Algorithm

We choose the Eisner algorithm as the graph search algorithm for our dependency parser. The Eisner algorithm searches a highest scoring tree given a sentence within the search space defined in 2.1. The score of a dependency tree given by $Score(x, y, \theta)$ on a graph level is decomposed into scores of subgraphs. Subgraphs are categorized into different orders according to the maximum number of nodes sharing the same head, as shown in Figure 2.3.

Eisner algorithm proposed by [10] is a greedy graph search algorithm which searches for a highest scoring projective tree of which the score is factorized into scores of first order subgraphs. In other words, the score of a tree is the sum of scores of individual arcs.

$$Score(x, y, \theta) = \sum_i score(x_i, y_{i1}, y_{i2}, \theta) \tag{2.2}$$

Given the scores of all possible arcs in the graph, Eisner algorithm ensures to find the highest scoring tree within the search space defined in Section 2.1 with $O(n^3)$ time where n is the length of the sentence. Eisner Algorithm constructs a dynamic programming table C[s][t][d][c] of size $n \times n \times 2 \times 2$ to keep track of the highest score of a subgraph which spans $x_s x_{s+1} ... x_t$ where $s < t$, d indexes the direction of the span(0=left,1=right) and c indexes the state of the span (0=complete,1=incomplete). The indices indicate four types of spans in the Eisner algorithm, right complete span, left complete span, right incomplete span, and left incomplete span. A right complete span take x_s as the head node of the span, x_t as the end point of the span and vice versa for a left complete span. A right incomplete span take x_s as the head node of the span, x_t as the target node of x_s and vice versa for a left incomplete

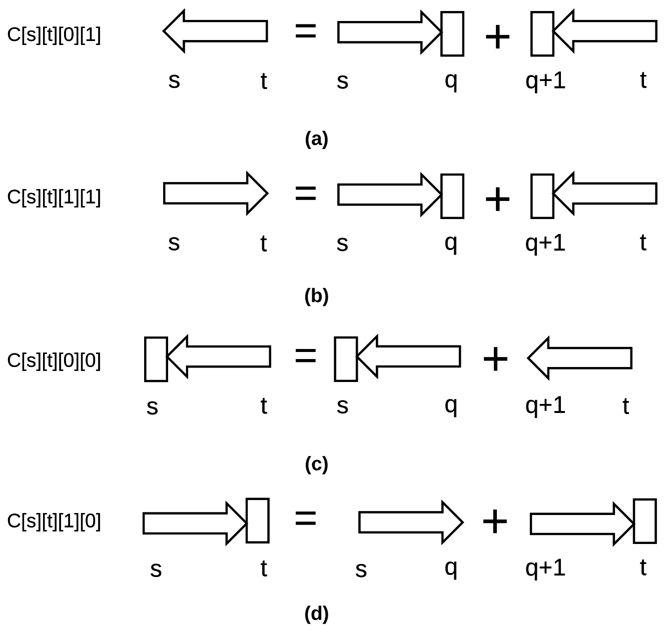


Figure 2.4: Graph Representation of Eisner Algorithm

span. In a complete span the end point is not necessary to be the target node of the head of the span. The Eisner algorithm starts with taking each single node as a left complete span or a right complete span with score 0, iteratively merging two spans until a right complete span from x_0 to x_n is reached.

Rules of joining spans are demonstrated in Figure 2.4. A complete span is graphically represented by $\rightarrow$ | or | $\leftarrow$. A incomplete span is graphically represented by $\rightarrow$ or $\leftarrow$. A left incomplete span and a right incomplete span can be joined to form a complete left or right span. A left complete span and a left incomplete span can be joined to form a left complete span. A right complete span and a right incomplete span can be joined to form a right incomplete span.

The highest score of a projective tree is stored in C[0][n][1][0]. As mentioned C[s][t][d][c] is a dynamic programming table, once C[0][n][1][0] is obtained one can trace back its trajectory to get the desired tree. Algorithm 1 [32] gives the standard Eisner algorithm.

2.3 Feedforward Neural Network

The feedforward neural network works as the score function in our dependency parser. This section only provides background knowledge of neural networks as well as optimization and regularization methods applied in our neural network model. In Section 3.3, we will introduce the neural network in the context of our dependency parsing problem.

The logic of human brain has been studied for years. Biological neurons in the brain are connected by dendrites and axons. A neuron accumulates signals received from dendrites and releases a impulse to the next neuron through an axon if the accumulated signal is upon a threshold. Inspired by biological neurons, researchers created feedforward neural network models to mimic the function of dendrites and axons. Neurons in a feedforward neural network model are hierarchically arranged by layers, namely input layer, hidden layer and output layer. Neurons in the input layer directly outputs system inputs to neurons in a hidden layer. The number of hidden layers may vary. Neurons in a hidden layer sums up inputs

Algorithm 1 Eisner Algorithm

```
0: input a sentence $x_0 x_1 ... x_n$, score(target $x_i$, head $x_j$, dependency type $r$, weight $\theta$)
1: Instantiate C[n][n][2][2]
2: for all s,d,c set C[s][s][d][c] = 0.0
3: for m in 1:n
4:     for s in 1:n
5:       t=s+m
6:       if t > n break
7:       $C[s][t][0][1] = max_{s \leqslant q < t}(C[s][q][1][0] + C[q+1][t][0][0]) + max_r score(x_s, x_t, r, \theta)$
         %subgraph a in Figure 2.4
8:       $C[s][t][1][1] = max_{s \leqslant q < t}(C[s][q][1][0] + C[q+1][t][0][0]) + max_r score(x_t, x_s, r, \theta)$
         %subgraph b in Figure 2.4
9:       $C[s][t][0][0] = max_{s \leqslant q < t}(C[s][q][0][0] + C[s][q][0][1])$
         %subgraph c in Figure 2.4
10:      $C[s][t][1][0] = max_{s \leqslant q < t}(C[s][q][1][1] + C[s][q][1][0])$
         %subgraph d in Figure 2.4
```

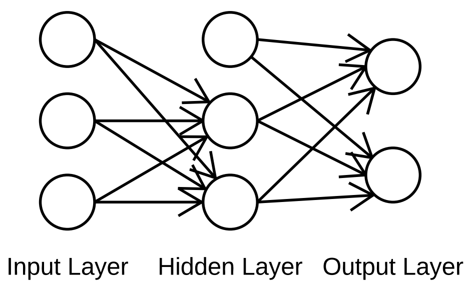


Figure 2.5: A Simple Example of Neural Networks

from neurons in the preceding layer and output the transformed summation to neurons in the succeeding layer. Finally, the output layer receives signals from the last hidden layer and produces the system output. Figure 2.5 gives a simple example of neural network models. Except the output layer, there is a bias neuron in each layer which does not connect neurons in the preceding layer. The bias neuron always outputs a value of 1. The use of the bias neuron is to push the plane away from origin.

In fact, the neural network is a generalized linear model with self-adaptive basis functions. It projects the raw data input into another representative feature space and builds a linear model upon transformed features. The feedforward neural network takes the form:

$$\begin{aligned} y(x,\theta) &= f(\theta_{n-1}^T \phi_{n-1}(x) + b_{n-1}) \\ \phi_{n-1}(x) &= g(\theta_{n-2}^T \phi_{n-2}(x) + b_{n-2}) \\ &\cdots \\ \phi_1(x) &= g(\theta_0^T \phi_0(x) + b_0) \\ \phi_0(x) &= x \end{aligned} \tag{2.3}$$

where $f(\cdot)$ is a nonlinear transformation in the case of classification and is an identical transformation in the case of regression [33]. The basis functions $\phi_i(\cdot)$ in layer i is itself a function of linear combination of basis functions in layer i-1 where $g(\cdot)$ is called the activation function. The parameter θ_i is the weight matrix from layer i to layer i+1 of size $n_i \times n_{i+1}$ where n_i is the number of neurons in the i^th layer.

2.3.1 Back Propagation Algorithm

The standard method of training neural networks is the back propagation algorithm [21]. It is a gradient descent method combined with chain rules. A neural network first receives an input signal and feed forward the signal down to the output layer. The output layer returns back an error term based on a loss incurred between the resulting signal and the desired output. The neural network system back propagates this error term to front layers iteratively. Algorithm 2 illustrates the back propagation algorithm. Let (p_m, q_m) denote the m^{th} training example, p_m is the initial input, q_m is the desired output, n denote the number of layers including the input layer, $l(\cdot)$ denote a given loss function. In the feedforward phase, after neurons being activated, the bias weight is added to the output vector in that layer. In the backpropagate phase, the error term does not include the bias neuron since it is not connected with its preceding layer. The weight of the bias neuron is excluded when calculating the error term.

Algorithm 2 Backpropagation Algorithm

```
0: **input** a training instance$(p_m, q_m)$
**phase1:feedforward**
1: $z_0 = p_m$
2: $a_0 = g_0(z_0)$
3: $for(i = 1; i < n; i++)\{$
4:   $a_{i-1} = concatenate(1, a_{i-1})$
5:   $z_i = \theta_{i-1}^T \times a_{i-1}$
6:   $a_i = g_i(z_i)$
7: $\}$
**phase2:backpropagate**
8:  $cost = l(q_m, a_{n-1})$
9:  $\delta_{n-1} = \frac{\partial l}{\partial z_{n-1}} = \frac{\partial l}{\partial a_{n-1}} \frac{\partial a_{n-1}}{\partial z_{n-1}} = l'(q_m, a_n - 1) \cdot g'_{n-1}(z_{n-1})$
10: $for(i = n-2; i >= 0; i--)\{$
11:   $\Delta\theta_i = \frac{\partial l}{\partial z_{i+1}} \frac{\partial z_{i+1}}{\partial \theta_i} = a_i \times \delta_{i+1}^T$
12:   $\delta_i = \frac{\partial l}{\partial z_i} = \frac{\partial l}{\partial z_{i+1}} \frac{\partial z_{i+1}}{\partial a_i} \frac{\partial a_i}{\partial z_i} = \theta_i[1 : end, 0 : end] \times \delta_{i+1} \cdot g'_i(z_i)$
% “end” refers to the index of the last row or column in the matrix.
13: $\}$
**phase3:update**
14: $\Delta\theta_i = \theta_i - \eta\Delta\theta_i$
```

Neural networks are universal approximators [23]. With the increase of hidden units, the risk of overfitting increases. Regularization methods are used to alleviate the problem of overfitting in statistical models. This thesis adopted two regularization methods, L2 regularization and Dropout. We will describe these two methods in the next two sections.

2.3.2 $L2$ Regularization

$L2$ regularization is a convenient method to regularize parameters. It adds a penalty term $0.5\lambda\theta^T\theta$ to the objective function. In a single step, it shrinks the weight matrix proportionally before moving against gradients. Overall it decays parameters of unimportant variables more than predicative variables. From the perspective of Bayesian statistics, given a MAP estimation problem, by applying $L2$ regularization it assumes a Gaussian prior for θ. For example, in linear regression, $L2$ regularization is known as ridge regression. The likelihood of a linear regression model is

$$p(D|\theta) = \prod_{i=1}^{N} N(y_i|\theta_0 + \theta^T x_i, \sigma^2) \tag{2.4}$$

Assuming the prior of θ is a Gaussian distribution with zero mean,

$$p(\theta) = \prod_j N(\theta_j|0, \frac{\sigma^2}{\lambda}) \tag{2.5}$$

The MAP estimates of the linear regression is

$$argmax_\theta p(D|\theta)p(\theta) \tag{2.6}$$

By taking logarithm and reversing the sign, Equation 2.6 is equivalent to minimizing

$$c(\theta) = \frac{1}{N}\sum_{i=1}^{N}(y_i - (\theta_0 + \theta^T x_i))^2 + \lambda||\theta||^2 \tag{2.7}$$

2.3.3 Dropout

Dropout proposed by Hinton [25] is effective to reduce generalization errors for neural network models. The dropout method thins a fully connected neural network by temporarily dropping out randomly chosen input and hidden neurons. In each iteration, a non-output layer neuron is present with probability p. The neuron being absent temporarily loses connections with its preceding and succeeding neurons while parameters associated with that neuron are retained to a next training iteration. The process of feedforward and backpropagation will not involve those absent neurons. Dropout actually samples a neural network model from a population of 2^n possible models where n is the number of non-output layer neurons. In test time, the proper way is to average predictions of all sampled neural networks. However, it is computationally expensive. The dropout method approximates the average of predictions by using a single fully connected neural network in test time. The output value of each neuron is multiplied by p so that the expected output value of a neuron is the same as that in the training process.

The dropout process added into the back propagation algorithm 2 is described by Equation 2.8 2.9 2.10.

In the feedforward phase,

$$\begin{aligned} z_i &= \theta_{i-1} \times a_{i-1} \\ a_i &= g_i(z_i) \\ r_i &\sim Bernoulli(p) \\ a_i &= a_i \cdot r_i \\ a_i &= concatenate(1, a_i) \end{aligned} \tag{2.8}$$

In the backpropate phase,

$$\begin{aligned} \delta_i &= \theta_i[1:end, 0:end] \times \delta_{i+1} \cdot g_i^{'}(z_i) \\ \delta_i &= \delta_i \cdot r_i \end{aligned} \tag{2.9}$$

In the prediction phase,

$$\begin{aligned} z_i &= \theta_{i-1}^T \times a_{i-1} \\ a_i &= g(z_i) \cdot p \end{aligned} \tag{2.10}$$

2.3.4 AdaGrad

AdaGrad is a variant of gradient descent method. This thesis used AdaGrad as the rule for updating parameters. An universal learning rate for updating parameters is not an optimal strategy. During training some parameters overshoot frequently while some parameters already become stable. Whether decreasing or increasing learning rate will not satisfy all parameters at the same time. Duchi and et.al [24] proposed an adaptive sub-gradient method called AdaGrad to adjust learning rate for parameters separately. Their method is based on an online learning setting. In online learning, a given loss function is not fixed because training data arrives through time. Without observing the whole dataset, parameters are updated in terms of a small regret. The regret of an online learning problem at time T is the accumulated excess loss if it does not train a model with the whole accumulated dataset. Let $l_t()$ denote the loss function in time t, θ_t denote the vector parameter estimates in time t and Θ denote the convex parameter space. The regret is expressed in equation below,

$$R(T) = \sum_{t=1}^{T} l_t(\theta_t) - inf_{\theta \in \Theta} \sum_{t=1}^{T} l(\theta) \tag{2.11}$$

Let η denote the learning rate, g_t denote the vector of gradients at time step T and $< a, b >$ denote the inner product of a and b. $G_t = \sum_{k=0}^{T} g_k g_k^{'}$. AdaGrad method employs the following updating rule:

$$\theta_{t+1} = argmin_{\theta \in \Theta} < \theta_t - \eta G_t^{-\frac{1}{2}} g_t, G_t^{-\frac{1}{2}} (\theta_t - \eta G_t^{-\frac{1}{2}} g_t) > \tag{2.12}$$

In implementation, we assume $\theta = \theta_t - \eta diag(G_t)^{-\frac{1}{2}} g_t \in \Theta$. To to avoid zero inversion we also add a $epsilon = 10^{-8}$ in the diagonal matrix. The updating rule becomes

$$\theta_{t+1} = \theta_t - \eta_t g_t \tag{2.13}$$

where $\eta_t = \eta diag(G_t + eps)^{-\frac{1}{2}}$

The adaptive learning rates η_t incorporate the history updating information of parameters. If a parameter oscillates often, the learning rate of that parameter will be shrank down. If a parameter is seldom updated because of data sparsity, once updated the learning rate of that parameter will still remain high. The drawback of AdaGrad is that adaptive learning rates are monotonically decreasing. If a learner jumps out of a local minimum, learning rates can not be lifted up again.

3 Methodology

This Chapter aims at explaining the way of training the score function in Equation 2.1 which is in the form of a neural network. Section 3.1 describes the training objective both from a probabilistic view and a non-probabilistic view as both perperpectives have good interpretations. Section 3.2 proposes the Loss-Augmented Eisner Algorithm that will be used in decoding a possible wrong tree with a high score during training. Section 3.3 introduces a neural network as a generative neural network and demonstrates how the neural network works as a score function. In this Chapter, we denote x^j as the j^{th} sentence in a training set and y^j as the desired output for the j^{th} sentence.

3.1 Max-margin Training Principle

Structured prediction can be seen as classifying an object to a possible structure out of an exponential large number of candidates. For example, in dependency parsing, it classifies a sentence x to a possible syntactic structure $\hat{y} \in Y$ where Y is a finite set. A big difference between a structured prediction problem and a common classification problem is that the output space Y is dependent on x for structured classification problems while the output space Y, such as $\{0, 1\}$, is fixed for classification problems. In dependency parsing, each sentence from a training set is different thus the output space of each sentence also differs. Assuming a distribution of y given data and parameters directly is not feasible for structured learning. However, since the output space is a finite discrete set, it is possible to assume the conditional distribution indirectly by Bayes rule.

$$p(y|x,\theta) = \frac{p(x,y|\theta)}{\sum_{y' \in Y} p(x,y'|\theta)} \tag{3.1}$$

Probabilistic approach in dependency parsing commonly assumes the distribution of (x, y) is from an exponential family, as shown in Equation 3.2.

$$p(x,y|\theta) \propto e^{\theta^T \phi(x,y)} \tag{3.2}$$

Let us assume a prior for θ,

$$p(\theta) = \frac{e^{-f(\theta)}}{Z} \tag{3.3}$$

where $f(\cdot)$ is a function of θ and Z is a constant. A Bayesian approach normally simulates the posterior distribution of parameters through MCMC. By Bayesian decision theory, the choice of parameters estimates is based on minimizing the posterior expected loss. For example, 0-1 loss corresponds to MAP estimates, quadratic loss corresponds to mean estimates, and absolute loss corresponds to median estimates. In our problem, a full Bayesian approach is intractable because of the number of parameters, the size of output space and the size of a training set. Max-margin training principle, in some literature referred as structural supporter machine, practically defines the posterior expected loss on a training set as a function of θ as shown in Equation 3.4. This function has no exact form however has been found an exact form for its upper bound. Therefore a set of point estimates can be computationally efficiently derived by minimizing the upper bound of posterior expected loss on a training set. By Murphy [34], the posterior expected loss on a training set is defined as

$$R_{EL}(\theta) = -logp(\theta) + \sum_{j=1}^{N} log(\sum_{y^j \in Y^j} L(y^j, \hat{y}^j) p(y^j | x^j, \theta)) \tag{3.4}$$

where $L(y^j, \hat{y}^j)$ is the loss incurred when the predicted structure is $\hat{y}^j$ and the true structure is y^j. Set $L(y^j, \hat{y}^j)$ to be the exponential of the empirical loss $\Delta(y^j, \hat{y}^j)$. In the other way around, $\Delta(y^j, \hat{y}^j) = logL(y^j, \hat{y}^j)$. By Murphy [34], the upper bound of posterior expected loss is

$$R_{EL}(\theta) \leqslant f(\theta) + \sum max_{\hat{y}^j \in Y^j} (\theta^T \phi(x^j, \hat{y}^j) + \Delta(y^j, \hat{y}^j)) - \theta^T \phi(x^j, y^j) \tag{3.5}$$

Max-margin training principle derives its name from a non-probabilistic view. It maximizes the margin between a correctly predicted structure and an incorrectly predicted structure. The goodness of a possible structure is measured by a score function $Score(x, y, \theta)$ interpreted as the log joint probability from the probabilistic view.

$$Score(x, y, \theta) = \theta^T \phi(x, y) \tag{3.6}$$

The training criteria is that the excess score between a correct structure and an incorrect structure should be at least larger than a margin loss $\Delta(y, \hat{y})$.

$$Score(x, y, \theta) - Score(x, \hat{y}, \theta) \geqslant \Delta(y, \hat{y}) \tag{3.7}$$

By Ratliff [35], Equation 3.7 leads to minimize the following objective function

$$c(\theta) = \frac{1}{n} \sum (l_j(\theta) + \frac{\lambda}{2} ||\theta||^2) \tag{3.8}$$

where $c(\cdot)$ represents the objective function, $l_j(\theta) = max_{\hat{y}^j \in Y^j}(Score(x^j, \hat{y}^j, \theta) + \Delta(y^j, \hat{y}^j)) - Score(x^j, y^j, \theta)$

In Equation 3.5, if the prior of θ is assumed to be $N(0, \lambda^{-1} I_n)$, then $f(\theta) = 0.5\lambda||\theta||^2$. The upper bound of the posterior expected loss on a training set has the same form as the objective function in Equation 3.8. The empirical loss $\Delta(y^j, \hat{y}^j)$ in Equation 3.4 corresponds to the margin loss in Equation 3.7 . In dependency parsing, the margin loss can be the number of incorrect predicted arcs. Since $\Delta(y^j, \hat{y}^j)$ is not differentiable with respect to θ, the sub-gradient method is used. The Sub-gradient method first introduced by Naum [36] deals with the problem of optimizing non-differentiable convex functions. In a gradient descent algorithm, one can update parameters by sub-gradients instead of gradients. A sub-gradient of convex function f is any vector g satisfying the inequality,

$$f(\theta_1) \geqslant f(\theta_2) + g(\theta_1 - \theta_2) \tag{3.9}$$

By Equation 3.9, the sub-gradient of the loss function $l_j(\theta)$ is

$$\frac{\partial}{\partial \theta} l_j(\theta) = \frac{\partial}{\partial \theta} Score(x^j, \hat{y}^j, \theta) - \frac{\partial}{\partial \theta} Score(x^j, y^j, \theta) \tag{3.10}$$

where $\hat{y}^j = argmax_{\hat{y}^j \in Y^j}(Score(x, \hat{y}^j, \theta) + \Delta(y^j, \hat{y}^j))$

The derivation of $\hat{y}^j$ is a result of a graph search algorithm. In next section, we will propose the Loss-Augmented Eisner Algorithm for decoding $\hat{y}$ in Equation 3.10.

3.2 Loss-Augmented Eisner Algorithm

The Eisner Algorithm decomposes the score of a tree into scores of individual arcs.

$$Score(x, y, \theta) = \sum_i score(x_i, y_{i1}, y_{i2}, \theta) \tag{3.11}$$

Taking the probabilistic view from the max-margin training objective, a score function is interpreted as the log joint probability of (x, y). This allows us to interpret Equation 3.11 as assuming the occurrence of an arc is independent of other arcs. The Eisner algorithm is a combinatorial optimization algorithm which searches for a highest scoring tree given scores of all possible arcs. However, in the max-margin training objective, one necessary step is to search for $\hat{y}^j = argmax_{\hat{y}^j \in Y^j}(Score(x, \hat{y}^j, \theta) + \Delta(y^j, \hat{y}^j))$. It is the highest scoring tree taking the margin loss $\Delta(y^j, \hat{y})$ into account. The margin loss $\Delta(y^j, \hat{y})$ is defined as the number of incorrectly predicted arcs discounted by a factor τ [5] [6],

$$\Delta(y^j, \hat{y}^j) = \tau \sum_i (\hat{y}^j_{i1} \neq y^j_{i1} || \hat{y}^j_{i2} \neq y^j_{i2}) \tag{3.12}$$

The parameter τ controls the scale of the margin loss. During training, a margin loss let a graph search algorithm find an wronger tree with a high score. By the training objective, the score of this wrong tree will be decreased and the score of the correct tree will be increased. If the parameter τ is too small, the returned tree by a graph search algorithm might be close to the correct tree. Scores of two trees in the training objective mostly canceled each other. If the parameter τ is too large, the returned tree will be far from the correct tree. However, the returned tree might have a low score. Updating parameters by the direction of an incorrect tree with a low score does not contribute to model performance as this incorrect tree is already unlikely to be outputted in prediction.

To solve the problem of finding a highest scoring tree taking margin loss into account, we propose the Loss-augmented Eisner Algorithm. To our knowledge, we are the first who incorporate the margin loss into the Eisner algorithm. In Pei et al. [5] and Kiperwasser et al. [6], the way they derive $\hat{y}$ in the max-margin training objective is not clear. The margin loss is integrated into the Eisner algorithm as an extra score τ when an arc does not exist in the correct tree as what Equation 3.12 indicates. Concretely, if an possible arc $(x^j_i, \hat{y}^j_{i1}, \hat{y}^j_{i2})$ in a sentence meet the condition $\hat{y}^j_{i1} \neq y^j_{i1} || \hat{y}^j_{i2} \neq y^j_{i2}$, the score of this arc will be added by τ. Algorithm 3 presents the Augmented-loss Eisner algorithm. Same steps as in Algorithm 1 is omitted.

3.3 Generative Neural Network

Feedforward neural networks are mostly applied as discriminative models. They model the conditional probability of y given data and parameters directly. In this section, we will present a feedforward neural network model as a generative model which works as a score function in a dependency parser. In dependency parsing, the generative neural network models the joint probability of a sentence and its syntactic structure. Assuming the occurrence of an arc is independent of other arcs in the sentence, the joint probability $p(x, y)$ is factorized as,

$$p(x, y) = \prod_{i=1}^{n} p(x_i, y_i) \tag{3.13}$$

Algorithm 3 Loss-augmented Eisner Algorithm

```
0: input a tree $(x, y)$, score(target $x_i$, head $x_j$, dependency type $r$, weight $\theta$)
...
6: if t > n break
7:   if $arc(x_s, x_t, argmax_r score(x_s, x_t, r, \theta)$ in the input tree$(x, y)$
8:      $C[s][t][0][1] = max_{s \leqslant q < t}(C[s][q][1][0] + C[q+1][t][0][0]) + max_r score(x_s, x_t, r, \theta)$
9:   else
10:     $C[s][t][0][1] = max_{s \leqslant q < t}(C[s][q][1][0] + C[q+1][t][0][0]) + max_r score(x_s, x_t, r, \theta) + \tau$
11:  if $arc(x_t, x_s, argmax_r score(x_t, x_s, r, \theta)$ in the input tree$(x, y)$
12:     $C[s][t][1][1] = max_{s \leqslant q < t}(C[s][q][1][0] + C[s][q+1][0][0]) + max_r score(x_t, x_s, r, \theta)$
13:  else
14:     $C[s][t][1][1] = max_{s \leqslant q < t}(C[s][q][1][0] + C[q+1][t][0][0]) + max_r score(x_t, x_s, r, \theta) + \tau$
15:  $C[s][t][0][0] = max_{s \leqslant q < t}(C[s][q][0][0] + C[s][q][0][1])$
16:  $C[s][t][1][0] = max_{s \leqslant q < t}(C[s][q][1][1] + C[s][q][1][0])$
```

Note that x_i represents the i^{th} word in the sentence, y_i consists of y_{i1} and y_{i2}, representing the head of x_i and the index of dependency type between x_i and y_{i1} respectively. Assume the joint distribution of (x_i, y_i) is from an exponential family, the generative neural network takes the form:

$$\begin{aligned} logp(x_i, y_i) &\propto \theta_{n-1}^T \phi_{n-1}(x_i, y_i) + b_{n-1} \\ \phi_{n-1}(x_i, y_i) &= g(\theta_{n-2}^T \phi_{n-2}(x_i, y_i) + b_{n-2}) \\ &\cdots \\ \phi_1(x_i, y_i) &= g(\theta_0^T \phi_0(x_i, y_i) + b_0) \end{aligned} \tag{3.14}$$

where θ_i is the weight matrix from layer i to layer $i+1$, $g(\cdot)$ is the activation function in a neural network and $\phi_i(\cdot)$ is the basis function in layer i. By convention let $Score(x, y)$ denote the log joint probability of x and y. Let $score(x, y)$ denote the joint probability of x_i and y_i

$$\begin{aligned} \because logp(x, y) &= \sum_i logp(x_i, y_i) \\ \therefore Score(x, y) &= \sum_i score(x_i, y_i) \end{aligned} \tag{3.15}$$

Replacing Equation 3.15 into the max-margin training objective in Equation 3.8 results in a neural network that minimizes the upper bound of posterior expected loss on a training set assuming Gaussian prior. The training objective of the neural network is to minimize

$$c(\theta) = \frac{1}{n} \sum_j max_{\hat{y}^j \in Y^j} (\sum_i score(x_i^j, \hat{y}_i{}^j, \theta) + \Delta(y^j, \hat{y}^j)) - \sum_i score(x_i^j, y_i^j, \theta) + \frac{\lambda}{2} ||\theta||^2 \tag{3.16}$$

There are two ways to get the score of an labeled arc (x_i, y_i) through a neural network model. One way is to use a basis function $\phi_0(x_i, y_{i1}, y_{i2})$ which encodes features about a labeled arc as an input and set one output neuron in the output layer. The score of the arc (x_i, y_i) equals to the output value of the neuron in the output layer. Another way is to use a basis function $\phi_0(x_i, y_{i1})$ which encodes features about an unlabeled arc as an input and set the number of output neurons equal to the number of dependency types. The score of the arc (x_i, y_i) equals to the output value of the ${y_{i2}}^{\text{th}}$ neuron in the output layer. We choose the latter approach in this thesis. The latter approach is computationally efficient in prediction as it simultaneously calculates scores of an arc $\phi_0(x_i, y_{i1})$ with different labels. In Section 3.3.1, we will define the model input $\phi_0(x_i, y_{i1})$. In Section 3.3.2, we will introduce activation functions used in this thesis. In Section 3.3.3, we will present our model architecture and validate that the model with $\phi_0(x_i, y_{i1})$ as an input is still a generative model.

3.3.1 Model Input

Word Embeddings Word embeddings are used for representing words in our model. Representing words by their index in a vocabulary creates millions of dummy variables. A single index uniquely identifies a word but contains no further information about the word. It generalizes poor and requires domain knowledge to add features of higher orders. Researchers in recent years have developed statistical models to learn word representations. Word representations, known as word embeddings, are vectors of real numbers mapped to words. Word embeddings can be regarded as basis functions. The notion of basis function occured in many statistical models. Basis functions in a kernel method map an original data point to a set of similarity measures between that data point and some reference points. Basis functions in a decision tree model project an original data point to decision regions. Similarly the mapping from a word to a vector of real numbers can be considered as an adaptive basis function for this word where the vector values are parameter estimates in terms of a learning objective. Word embeddings can be trained by one model and reused as initializations in another model. There are two prevailing models to obtain pre-trained word embeddings, word2vec and GloVe. The word2vec model proposed by Mikolov et al. [28] learns word embeddings through the objective of maximizing the average conditional probability of the occurrence of words within a window of an observed word. The GloVe model proposed by Pennington et al. [29] captures ratios of words co-occurrence probability. An interesting finding about wordvec and GloVe is that they can perform semantic and syntactic word analogy task by vector additions. For example, Figure 3.1 shows the projection of some GloVe word embeddings in 2 dimensions[2]. As it is seen in the graph, vec("woman")-vec("man")+vec("king") is closest to vec("queen"). Therefore the word "queen" can be deduced by vector additions. In this thesis, we obtained pre-trained GloVe word embeddings of 50 dimensions online[3]. It was trained with Wikepedia 2014 and Gigaword corpus.

Input The input to our model on an instance level is the feature representation of an arc. The basis function $\phi_0(\cdot)$ transforms an arc to its feature representation. Let k denote the index of word y_{i1} in the sentence and $\phi^w(\cdot)$ denote the basis function of a word, which is a dictionary mapping a word to its word embeddings, and $\phi^p(\cdot)$ denote the postag embeddings of a word which is a dictionary mapping a word to its postag embeddings. The postag embeddings similarly as word embeddings represent a postag by a vector of real values. The basis function of an arc (x_i, y_{i1}) in this thesis is defined as

$$\begin{aligned}\phi_0(x_i, y_{i1}) = &\mathit{concatenate}(\phi^w(x_{k-2}), \phi^w(x_{k-1}), \phi^w(x_k), \phi^w(x_{k+1}), \phi^w(x_{k+2}), \phi^w(x_{i-2}), \\ &\phi^w(x_{i-1}), \phi^w(x_i), \phi^w(x_{i+1}), \phi^w(x_{i+2}), \phi^p(x_{k-2}), \phi^p(x_{k-1}), \phi^p(x_k), \phi^p(x_{k+1}), \\ &\phi^p(x_{k+2}), \phi^p(x_{i-2}), \phi^p(x_{i-1}), \phi^p(x_i), \phi^p(x_{i+1}), \phi^p(x_{i+2}), \mathit{distance}(x_i, y_{i1}))\end{aligned} \tag{3.17}$$

where $\mathit{distance}(\cdot)$ measures the number of words in between x_i and y_{i1}. The larger the distance, the more unlikely two words will form an arc. For the purpose of a small scale, in our dependency parser the distance measure is defined as

$$\mathit{distance}(x_i, y_{i1}) = \begin{cases} 0.09, & \text{if } |i-k| > 10 \\ 0.06, & \text{if } 5 < |i-k| \leqslant 10 \\ 0.03, & \text{if } |i-k| = 5 \\ 0, & \text{if } |i-k| = 4 \\ -0.03, & \text{if } |i-k| = 3 \\ -0.06, & \text{if } |i-k| = 2 \\ -0.09, & \text{if } |i-k| = 1 \end{cases} \tag{3.18}$$

[2]excerpted from http://nlp.stanford.edu/projects/glove/

[3]http://nlp.stanford.edu/projects/glove/

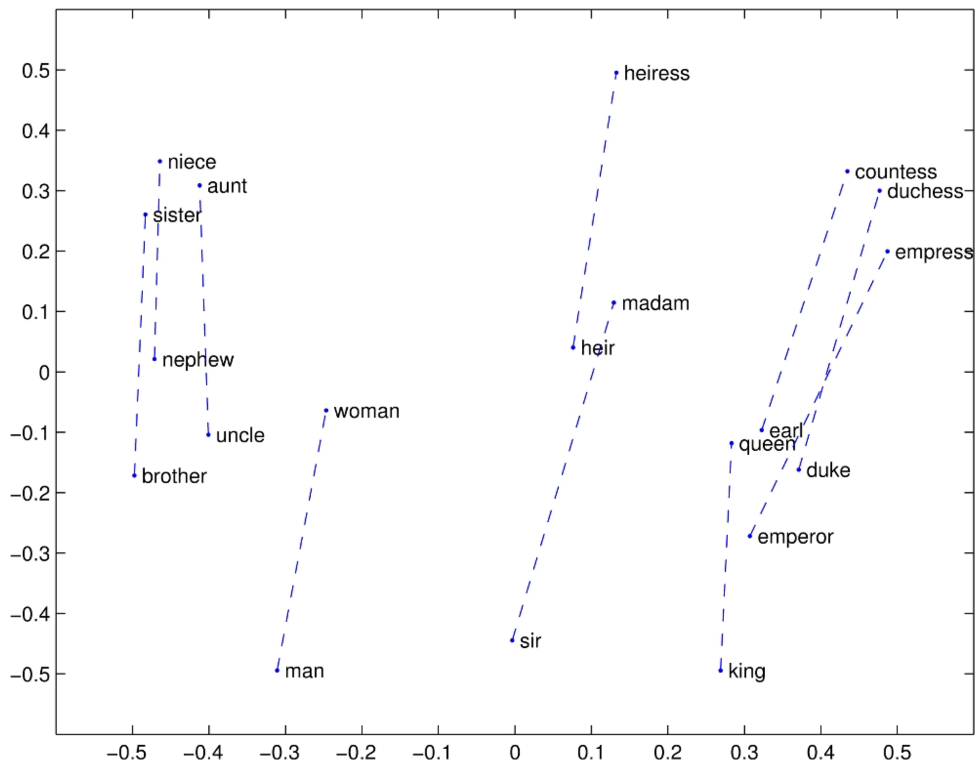


Figure 3.1: Visualization of GloVe Word Embeddings in 2 Dimensions

The basis function $\phi_0(\cdot)$ incorporates a window of neighboring word embeddings and postag embeddings of a head and a dependent. The size of the context window is actually a hyper parameter. A small context window does not contain enough information. A large context window introduces a bit more noises. Technically, we can experiment with different window sizes but for time issue we choose the window size as 2 in this thesis. For example, in the sentence,"Many investors certainly believe a bidding war is imminent", the context window for "a" is "certainly","believe","bidding" and "war" and the context window for the head of "a", which is "war", is "a","bidding","is","imminent". b

3.3.2 Activation Function

Activation functions play a key role in a neural network model. They enable the neural network to learn non-linear relations. We list three activation functions used in the thesis:

- tanh(tangent hyperbolic):
  $tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
- tanhcubic(tangent hyperbolic cubic):
  $tanhcubic(z) = tanh(z^3 + z)$
- cubic:
  $cubic(z) = z^3$

Sigmoid and tanh activation function are two common activation functions. The tanh activation function is preferred than the sigmoid activation function because the sigmoid activation function constrains outputs of neurons within positive values. It is the reason that we did not choose sigmoid activation function. The tangcubic function is a novel activation function proposed by Pei et al. [5]. Their experimental results showed that the tanhcubic function is

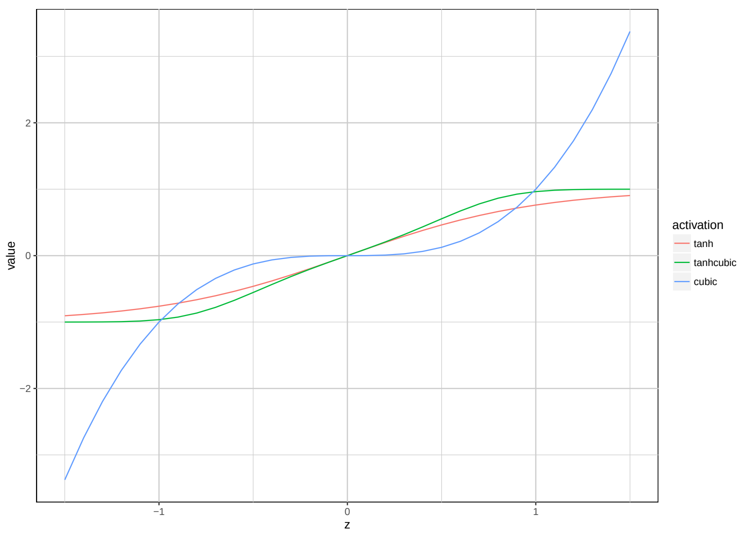


Figure 3.2: Plot of Activation Functions

more effective than the tanh and the cubic function. However, according to Figure 3.4, as the effective region of the tanhcubic function is narrower than the tanh function, it is doubted that the tanhcubic function has any advantage over the tanh function. Compared to the tanh and the tanhcubic function, the cubic function is an unbounded function. The cubic function showed a better performance in Chen et al.'s work [31]. We will experiment with these three activation functions in Section 4.6.

3.3.3 Model Architecture

Our model architecture is a feedforward neural network with one hidden layer and multiple output units as shown in Figure 3.3. The input of the neural network is the basis function $\phi_0(x_i, y_{i1})$ defined in Section 3.3.1. It mainly includes the word embeddings and the postag embeddings of a head, a dependent and their surrounding words within a window of 2. The neural network has one hidden layer and multiple output units. The number of hidden neurons by default is 200. The number of output units equals to the number of dependency types. The score of (x_i, y_i) is the $y_{i2}{}^{\text{th}}$ output of the output layer. For example, in Figure 1.1, the score of the arc pointing from "war" to "a" with dependency type NMOD, is the output value of the corresponding neuron labeled with NMOD as shown in Figure 3.3.

It seems that this architecture models $p(y_{i2}|x_i, y_{i1})$ since the basis function in the input layer is a function of (x_i,y_{i1}) and the output is a distribution over y_{i2}. However one can regard the basis function in Equation 3.14 $\phi_1(x_i, y_i)$ as $\phi_1^*(x_i, y_{i1}) \times \phi_r(y_{i2})$, where $\phi_1^*(x_i, y_{i1}) = g(\phi_0(x_i, y_{i1}) + b_0)$ and $\phi_r(y_{i2})$ is a row vector of which the $y_{i2}{}^{\text{th}}$ column is one and zero otherwise. To give a specific example, consider one instance with $y_{i2} = 2$ and suppose we have three hidden units, three output units. The score function $score(x_i, y_i)$ equals to

$$score(x_i, y_i) = \theta_1^T \times \phi_1(x_i, y_i) + b_1$$

$$\Leftrightarrow$$

$$\begin{bmatrix} \theta_1^{11} & \theta_1^{12} & \theta_1^{13} \\ \theta_1^{21} & \theta_1^{22} & \theta_1^{23} \\ \theta_1^{31} & \theta_1^{32} & \theta_1^{33} \end{bmatrix} \times \left(\begin{bmatrix} \phi_1^1 \\ \phi_1^2 \\ \phi_1^3 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \right) + b_1 = \begin{bmatrix} \theta_1^{11} & \theta_1^{12} & \theta_1^{13} \\ \theta_1^{21} & \theta_1^{22} & \theta_1^{23} \\ \theta_1^{31} & \theta_1^{32} & \theta_1^{33} \end{bmatrix} \times \begin{bmatrix} 0 & \phi_1^1 & 0 \\ 0 & \phi_1^2 & 0 \\ 0 & \phi_1^3 & 0 \end{bmatrix} + b_1 \quad (3.19)$$

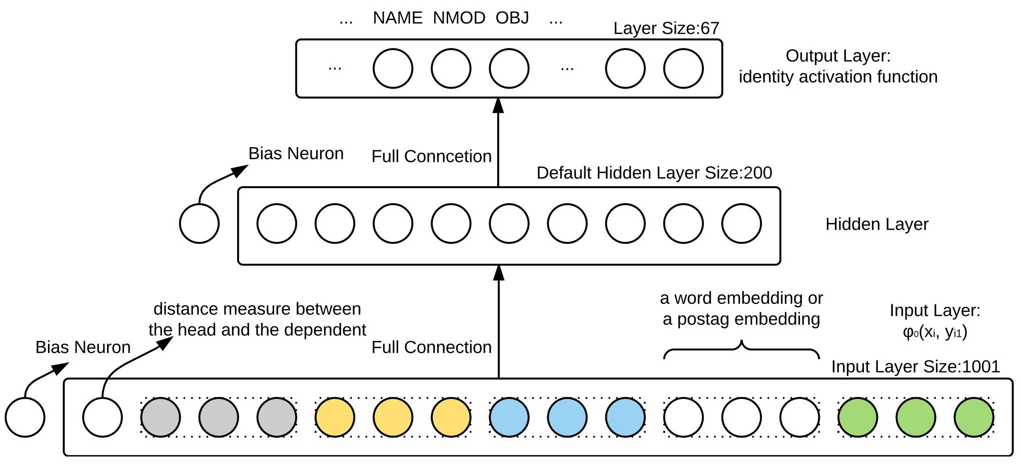


Figure 3.3: The Neural Network Architecture

If we reshape the two matrices on the right side of Equation 3.19 into vectors, we will get

$$\begin{aligned} \theta_1 &= [\theta_1^{11}; \theta_1^{12}; \theta_1^{13}; \theta_1^{21}; \theta_1^{22}; \theta_1^{23}; \theta_1^{13}; \theta_1^{23}; \theta_1^{33}] \\ \phi_1(x_i, y_i) &= [0; 0; 0; \phi_1^1; \phi_1^2; \phi_1^3; 0; 0; 0] \\ score(x_i, y_i) &= \theta_1^T \phi_1(x_i, y_i) = \theta_1^{21}\phi_1^1 + \theta_1^{22}\phi_1^2 + \theta_1^{23}\phi_1^3 + b_1 \end{aligned} \tag{3.20}$$

This example illustrates that our model with $\phi_0(x_i, y_{i1})$ as an input in fact incorporates information about y_{i2} into the model architecture. Therefore our model is still a generative model defined in the beginning of this Section.

Details of Implementation In implementation, we use the back-prorogation algorithm for training the neural network. Since the back-propagation algorithm is a gradient descent method which minimizes a training objective. It can be adjusted to minimize Equation 3.16 by customizing the loss function $l(\cdot)$ in Algorithm 2. Minimizing Equation 3.16 involves increasing scores of real arcs and decreasing scores of incorrectly predicted arcs. In the following, we will demonstrate how to increase the score of an arc (x_i, y_i) through an example. In the step 8 of Algorithm 2, the loss function $l(\cdot)$ takes two arguments the desired output q_m and the actual output a_{n-1}. Suppose $a_{n-1} = [0.1, 0.2, 0.3]^T$ and $y_{i2} = 2$. As $y_{i2} = 2$, the second output value in the output layer is supposed to be increased. We artificially design $q_m = [0, -1, 0]^T$. The loss function $l(\cdot)$ in Algorithm 2 is customized as

$$l(q_m, a_{n-1}) = q_m^T a_{n-1} \tag{3.21}$$

By Equation 3.21, in this example, $l(q_m, a_{n-1}) = -0.2$. As the back propagation is an gradient descent algorithm, decreasing -0.2 is equivalent to increasing 0.2. Alternatively, if we want to decrease the score of (x_i, y_i), we can artificially set $q_m = [0, 1, 0]^T$.

As Equation 3.21 is differentiable, the back propagation algorithm works exactly the same for the generative neural network. Noticeably, in Section 3.3.1, we mentioned the input of our neural network model is a self adaptive basis function. It indicates the word embeddings and postag embeddings are not fixed. They are treated as parameters in the model and updated in each iteration by its derivatives which are blocks of the input layer error term δ_0 in Algorithm 2.

3.4 Model Summary

In the end of this Chapter, we make a short summary of our model. The model assumptions made in this thesis are:

- The syntactic structure of sentences satisfy four properties of dependency trees.
- The syntactic structure of sentences are projective dependency trees.
- The probability of the occurrence of an arc is independent of other arcs in a dependency tree.

The training objective is

$$c(\theta) = \frac{1}{n}\sum_{j} max_{\hat{y}^j \in Y^j}(\sum_{i} score(x_i^j, \hat{y}_i{}^j, \theta) + \Delta(y^j, \hat{y}^j)) - \sum_{i} score(x_i^j, y_i^j, \theta) + \frac{\lambda}{2}||\theta||^2 \tag{3.22}$$

where $\Delta(y^j, \hat{y}^j) = \tau \sum_i (\hat{y}_{i1}^j \neq y_{i1}^j || \hat{y}_{i2}^j \neq y_{i2}^j)$, i.e., the number of incorrectly predicted dependency relations discounted by a factor. The score function is a neural network model with one hidden layer. The input of the neural network is $\phi_0(x_i, y_{i1})$. The number of neurons in the output layer equals to the number of dependency type. The score of an arc (x_i, y_{i1}) is the ${y_{i2}}^{\text{th}}$ output of the neural network. The Eisner algorithm is used for prediction and the Loss Augmented Eisner Algorithm is used for decoding $\hat{y}^j$ in Equation 3.22. Parameters in the model include weight matrices of the neural network, word embeddings, and postag embeddings. We trained the neural network by AdaGrad with mini-matches. Dropout is only applied to the hidden layer and L2 regularization is applied to parameters in the neural network, word embeddings, and postag embeddings.

To give readers a better understanding of how our dependency parser is trained, Figure 3.4 presents a draft of the work flow of a training process. For simplicity let us assume the batch size is one. During training, after one epoch is completed, the data will be shuffled randomly.

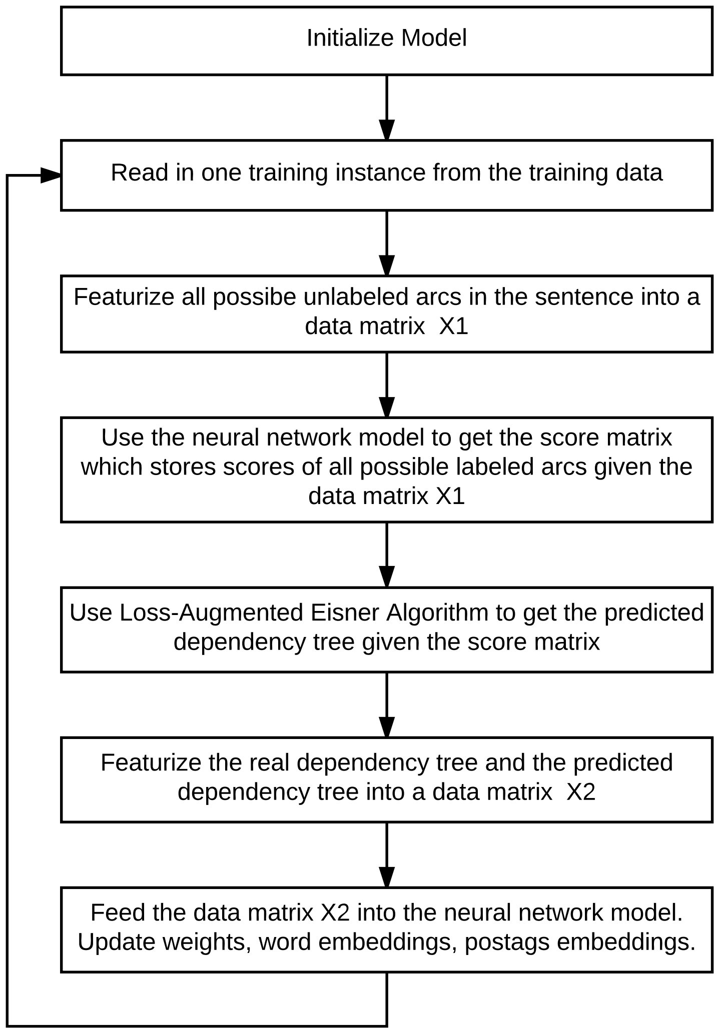

Initialize Model
Read in one training instance from the training data
Featurize all possibe unlabeled arcs in the sentence into a data matrix X1
Use the neural network model to get the score matrix which stores scores of all possible labeled arcs given the data matrix X1
Use Loss-Augmented Eisner Algorithm to get the predicted dependency tree given the score matrix
Featurize the real dependency tree and the predicted dependency tree into a data matrix X2
Feed the data matrix X2 into the neural network model. Update weights, word embeddings, postags embeddings.

Figure 3.4: The Work Flow of the Training Process

4 Experiments and Results

This chapter reports the main experimental results of this thesis. First we assessed characteristics of Penn treebank data and GloVe word embeddings. Second, we conducted several groups of experiments for parameter tuning. Next, we experimented with different initialization of word embeddings and postag embeddings. Finally, we experimented with different margin loss functions. The model with highest performance is selected among all experiments. We analysed predictions produced by this model. In the end, we compared our model with Pei et al.'s model.

4.1 Penn Treebank Data

We choose the Penn Treebank Data as the training data in this thesis. Penn treebank is a large annotated corpus created by University of Pennsylvania [37]. It contains a wide range of materials include IBM computer manuals, nursing notes, Wall Street Journal articles and so forth [38]. We used the Wall Street Jounal section(section 2-24) of Penn treebank which is a phrase-structured treebank. We used deterministic rules agreed on The Conference on Computational Natural Language Learning in 2008 (CoNLL 2008) [39] to convert phrase-structured Penn treebank into a dependency treebank. The dataset is split into three partitions by standard, sec2-21 as training set , sec22 as validation set and sec23 as test set. In the dataset, a sentence is decomposed of instances of words. Sentences are separated by a blank line. Table 4.1 provides a description of the fields of the dataset. Since values of field 3,4,5,6,9,10 are missing, they are not reported. Table 4.2 shows the data format about the graph representation of a dependency tree in Figure 1.1. One can reconstruct a depedency tree according to Table 4.2. For example, the head of the word "Many" is the 2$^{\text{nd}}$ word in the sentence which is "investors" and their dependency type is "NMOD".

4.2 Evaluation Metrics

We use Unlabeled Attachment Score (UAS) and Labeled Attachment Score (LAS) to measure the performance of a dependency parser. The single head property of dependency trees guarantees each word in the sentence only has one head. As the prediction for a word x_i is its single head and the dependency type between the word and its head, i.e. $(\hat{y}_{i1}, \hat{y}_{i2})$, the accu-

Field Number	Field Name	Description
1	ID	Index of a token, starting from 1 in each sentence
2	FORM	Token form
4	POSTAG	Part of speech tag
7	HEAD	Head ID of the current token, 0 refers to the root node
8	DEPREL	Dependency type of the arc between the current token and its head

Table 4.1: Fields Description

ID	FORM		POSTAG			HEAD	DEPREL		
1	Many	_	JJ	_	_	2	NMOD	_	_
2	investors	_	_NNS	_	_	4	SBJ	_	_
3	certainly	_	RB	_	_	4	ADV	_	_
4	believe	_	VBP	_	_	0	ROOT	_	_
5	a	_	DT	_	_	7	NMOD	_	_
6	bidding	_	JJ	_	_	7	NMOD	_	_
7	war	_	NN	_	_	8	OBJ	_	_
8	is	_	VBZ	_	_	4	OBJ	_	_
9	imminent	_	JJ	_	_	8	PRD	_	_
10	.	_	.	_	_	4	P	_	_

Table 4.2: A Instance of Dependency Treebank

racy of a dependency parser can be assessed in terms of unlabeled arcs $(x_i, \hat{y}_{i1})$ and labeled arcs $(x_i, \hat{y}_{i1}, \hat{y}_{i2})$. This is reflected by UAS and LAS in Equation 4.1 4.2.

$$\text{UAS} = \frac{\text{Number of Correctly Predicted Unlabeled Arcs}}{\text{Total Number of Arcs}} \tag{4.1}$$

$$\text{LAS} = \frac{\text{Number of Correctly Predicted Labeled Arcs}}{\text{Total Number of Arcs}} \tag{4.2}$$

To give an example, suppose there is only one training instance, “John saw Mary.”, in the dataset. The real dependency tree and the predicted dependency tree is shown in Table 4.3. In the example, there is two correctly predicted unlabeled arcs and one correctly predicted labeled arc. “Mary” is predicted with correct head but incorrect dependency type. “.” is predicted with both correct head and dependency type. The UAS is $2/4 = 0.5$ and the LAS is $1/4 = 0.25$. Correctly predicting heads of punctuations is not important. In this thesis, we exclude punctuations when calculating UAS and LAS.

4.3 Data Preprocessing

Data preprocessing is a neccessary procedure. In this thesis, it includes three steps:

- Lowercase all the words
- Replace words occurred only once in the training set as <UNKNOWN> word, and words in the validation set and test set absent in the training set as <UNKNOWN>

ID	FORM	HEAD	DEPREL	Predicted HEAD	Predicted DEPREL
1	John	2	SBJ	0	ROOT
2	saw	0	ROOT	1	SBJ
3	Mary	2	OBJ	2	NMOD
4	.	2	P	2	P

Table 4.3: An Example of Predictions

Statistics	Training Set	Validation Set	Test Set	# dependency types	# postags
# tokens	950348	40121	56702	67	45
# unique tokens	44352	6839	8423		
# sentences	39832	1700	2416		
average sentence length	23.86	23.60	23.47		

Table 4.4: Basic Statistics of Penn Treebank Data

word. Numbers of words replaced in the training set, validation set and test set are 17831, 1411 and 1809 respectively.

- Artificially set the two preceding tokens before the first word in the sentence as <START>, and the two succeeding tokens after the last word in the sentence as <END>. This is because the input of neural network contains word embeddings of surrounding words of the head and the dependent in an arc. It can be a problem if a token is the first node or the last node in the sentence so that it does not have 2 preceding tokens or 2 succeeding tokens.

4.4 Assessment of Data

In this section, we explore the nature of the Penn treebank data and the GloVe word embeddings. Table 4.4 summarizes basic statistics of Penn treebank data. The size of validation set and test set is about 5% of the training set. There is no obvious difference of average sentence length among three datasets. The unique number of dependency types and postag is 67, 45 respectively. Next we are interested in the distribution of dependency types. Figure 4.1 plots the frequency of dependency type in the training set. The top three most frequency dependency types are NMOD, P, PMOD. The number of root relation equals to the number of sentences as each sentence only has one root node. The number of subject relation is twice time as the number of root type, indicating on average each English sentences in the training set has one subordinate clause. The distribution of dependency types is quite unbalanced. We will analyze whether the nature of this unbalance will affect the parser prone to output dependency types which are of the majority in Section 4.9.

A neural network is trained faster if the scale of variables is of the same range. We investigate the scale of word embeddings in the training data. Figure 4.1 shows the boxplot of word embeddings with 50 dimensions. As it can be seen, the scale of the word embeddings is almost at the same range from -1 to 1 except the 31$^{\text{th}}$ dimension. We do not know the reason that this dimension is special as the data is abstract representations of words. Standardizing the word embeddings by minus mean and divided by standard deviation will to some extent lose the syntactic and semantic information kept in the word embeddings. We alternatively normalize word embeddings in unit length since the cosine similarity between two word vectors will not be changed by vector normalization. Figure 4.2b shows the boxplot after vec-

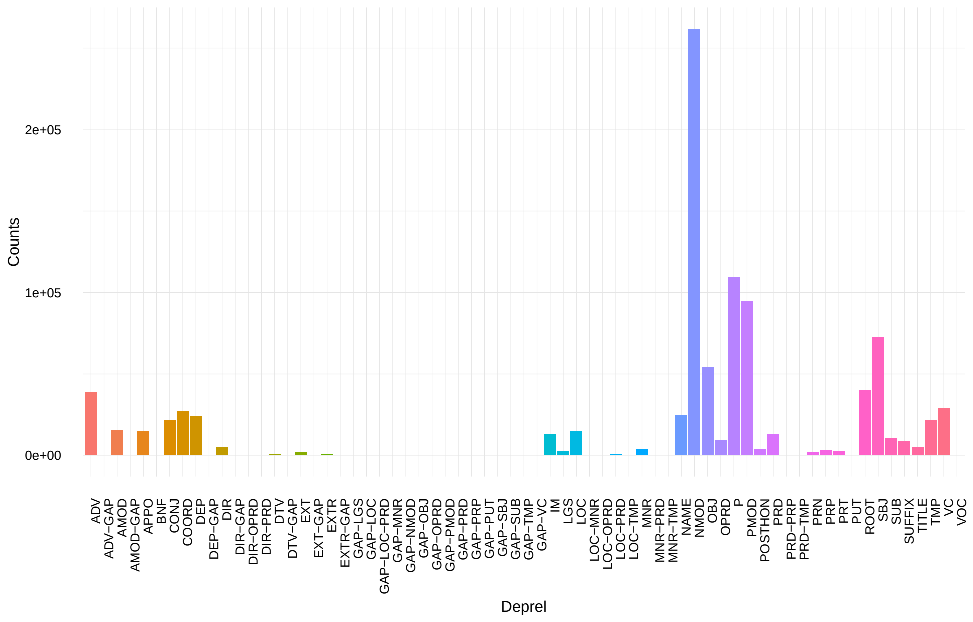
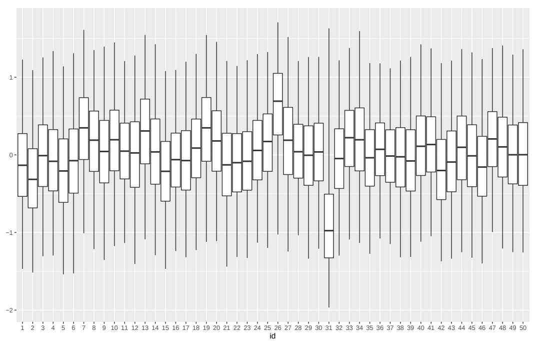
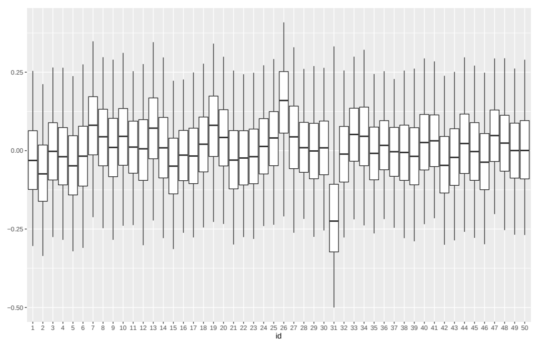


Figure 4.1: Barplot of Dependency Types



(a) before Normalization



(b) after Normalization

Figure 4.2: Boxplot of GloVe Word Embeddings

Quantile	2%	25%	50%	75%	98%
GloVe	-1.265	-0.367	0.020	0.407	1.316
normGloVe	-0.267	-0.085	0.005	0.095	0.277

Table 4.5: Comparison of Quantile Mean

tor normalization, it seems that the problem is not solved. However we can also randomly initialize word embeddings and postag embeddings to force the data in the same scale.

Table 4.5 calculated the averaged quantile among 50 dimensions for GloVe word embeddings and normalized Glove word embeddings. The motivation is to set the postag embedding around the same range with word embeddings. According to Table 4.5, we roughly designed three settings of data representations as shown in Table 4.5.

Embedding Setting	Word Embeddings	Postag Embeddings
Emb1	Glove	runif(-1,1)
Emb2	normGlove	runif(-0.2,0.2)
Emb3	runif(-0.01,0.01)	runif(-0.01,0.01)

Table 4.6: Embedding Settings

Running Time of Training	2.8 hour/epoch
Parsing Speed	13.9 sentence/second

Table 4.7: Table of Running Time

4.5 Computer Environment

The computing platform is iMAC with 2.5 GHz Intel Core i5 processor and 12 GB RAM. The dependency parsing system is implemented in JAVA with Gradle build manager. The library ND4j is used for matrix operations. The Eisner algorithm has already been created in an earlier system [4]. We adjust it to be compatible with the neural network. The neural network along with the training objective, dropout and AdaGrad were implemented by the author. Table 4.7 reports the average running time and the prediction speed of our dependency parser.

4.6 Hyper Parameter Tuning

Due to the non-linearity of neural networks, the objective function is non-convex. Therefore there are many local optimums. It is important to tune the hyper parameters to find a better local optimum. The hyper parameters in the neural network model include learning rate, number of hidden neurons, dropout rate, l2 rate λ, activation function, batch size, margin loss discount factor τ, the dimension of word embeddings, the dimension of postag embeddings. The total number of hyper parameters is 9. Suppose each hyper parameter has three options. A grid search will loop through $3^9 = 19683$ possibilities. Considering the running time of one epoch, it is impractical for us to explore the hyper parameter space exhaustively. Since parameter tuning is not the top priority of this thesis, we adopted a basic parameter setting from Chen et al. [31] and Pei et al. [5]. We did a preliminary exploration to see if the model performance is sensitive to any single change of hyper parameters. The basic settings of our model are word embedding size 50 randomly initialized from uniform distribution (-0.01,0.01), postag embedding size 50 randomly initialized from uniform distribution (-0.01,0.01), parameters of the neural network randomly initialized from uniform distribution (-0.01,0.01), hidden layer size 200, activation function tangent hyperbolic, AdaGrad learning rate 0.01, dropout rate 0.5, regularization parameter 0.0001, margin loss discount factor 0.3, batch size 20 and epoch 1. In the following experiments, we keep all parameters the same except the parameter to be tuned.

From Table 4.8, the learning rate is the most sensitive hyper parameter. A high learning late severely impairs the model performance. The choice of learning rate 0.01 is relative optimal. Tuning the number of hidden units, dropout rate, l2 rate, batch size, and margin loss discount factor does not give a significant improvement of model accuracy. The effect of these hyper parameters are not seen partly due to the fact that the neural network is trained with only one epoch for the sake of computation time.

In experiment 5, the cubic activation function learns faster than tanh and tanhcubic. This is contrary to Pei et al.'s finding [5]. In their experiments, tanhcubic performs dramantically

[4]https://github.com/liu-nlp/beta

Experiment	Random Seed	Learning Rate	UAS	LAS
1	7522	0.001	70.85%	64.64%
		0.01	85.86%	81.21%
		0.1	14.60%	2.81%
Experiment	Random Seed	Hidden Units	UAS	LAS
2	5438	100	85.47%	80.19%
		200	85.74%	80.74%
		400	85.57%	80.63%
Experiment	Random Seed	Dropout Rate	UAS	LAS
3	3794	0.5	85.84%	80.77%
		0.7	86.13%	80.96%
		1	85.89%	80.56%
Experiment	Random Seed	l2 Rate	UAS	LAS
4	2975	0.00001	85.94%	81.24%
		0.0001	85.85%	81.14%
		0.001	84.99%	79.90%
Experiment	Random Seed	Activation Function	UAS	LAS
5	8756	tanh	85.64%	80.82%
		tanhcubic	85.44%	80.60%
		cubic	87.59%	81.63%
Experiment	Random Seed	Batch Size	UAS	LAS
6	7534	20	85.72%	81.11%
		30	85.74%	80.83%
		40	85.29%	80.39%
Experiment	Seed	Margin Loss Discount Factor	UAS	LAS
7	2837	0.3	85.76%	81.16%
		0.6	85.80%	81.15%
		0.9	85.85%	81.04 %

Table 4.8: Hyper Parameter Tuning

better than tanh in the first epoch, as it will be shown in Figure 4.6a. In our opinion, Pei et al. used a bigger learning rate causing the cubic function unstable in the first several opochs as a neural network with the cubic activation function can explode with large inputs.

4.7 Experimental Result with Different Embedding Settings

In this experiment, our main interest is whether the pre-trained word embeddings improves the model performance. Three embedding settings were chosed in Section 4.4. The parameter settings are parameters of the neural network randomly initialized from uniform distribution (-0.01,0.01), hidden layer size 200, activation function cubic, AdaGrad learning rate 0.01, dropout rate 0.5, regularization parameter 0.0001, margin loss discount factor 0.3, batch size 20, and epoch 20 and random seed 3425. In the experiment, we keep all parameters the same except the word embeddings and postag embeddings.

Table 4.9 reports experimental results with three different embedding settings. From Table 4.9, Emb3 gives a better result than Emb1 and Emb2. Emb2 reached a same UAS as Emb1 but a lower LAS. Emb2 has a larger scale than Emb1 and Emb3 causing hidden neurons recieve larger inputs. As the cubic activation function increases inputs triply, the neural network with Emb2 might be unstable in the begining. Figure 4.3 shows the convergence plot of this experiment. The Unlabelled Attachment Scores of all three embeddings are converged. From Figure 4.3, Emb3 evidently learns faster than Emb1 and Emb2 in the first epoch. The Emb3

Experiment	Embedding Setting	Word Embeddings	Postag Embeddings	UAS	LAS
8	Emb1	runif(-0.01,0.01)	runif(-0.01,0.01)	91.44%	86.88%
	Emb2	glove	runif(-1,1)	91.44%	86.15%
	Emb3	normglove	runif(-0.2,0.2)	91.83%	87.11%

Table 4.9: Comparison of Embeddings Settings

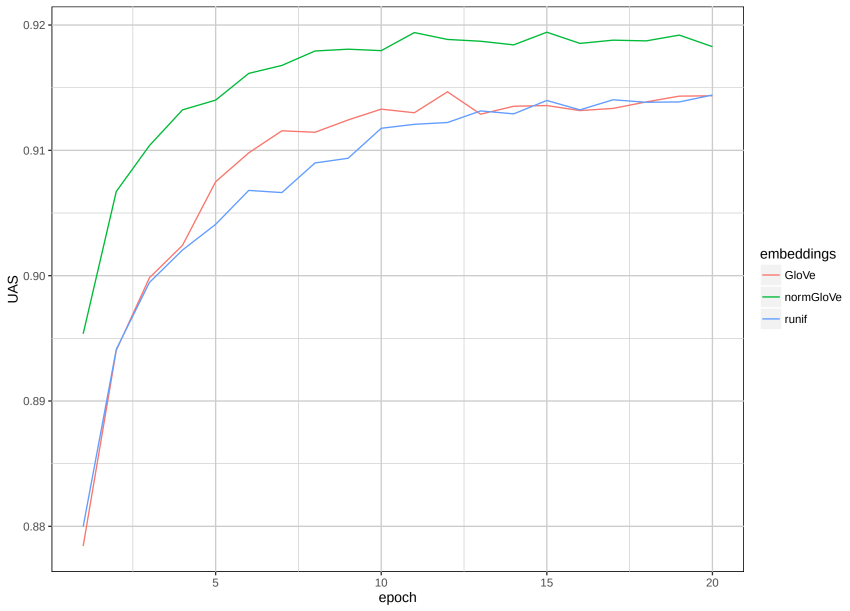


Figure 4.3: Convergence plot of Experiment 8

includes the normalized GloVe word embeddings. The normalized GloVe word embeddings retained cosine distance relationship among word embeddings. It is an effective initialization of word embeddings according to the results. In the next experiment, the embedding setting will be replaced by Emb3.

4.8 Experimental Results with Different Margin Loss Functions

In Section 3.1, we mentioned that the objective function can be interpreted as minimizing the upper bound of posterior expected loss. This allows us to adjust the margin loss $\Delta(y^j, \hat{y}^j)$ according to different assumptions. Previous experiments assumes that a loss is incurred if an arc is wrongly predicted both in terms of the head of a word and their dependency type. However, it is reasonable that we would expect a lower loss if the model predicts the right head of a word with a wrong dependency type. In this section, we compare different margin loss functions. These loss functions are defined as,

$$\Delta_1(y^j, \hat{y}^j) = 0 \tag{4.3}$$

$$\Delta_2(y^j, \hat{y}^j) = 0.3 \sum_i (\hat{y}^j_{i1} \neq y^j_{i1} || \hat{y}^j_{i2} \neq y^j_{i2}) \tag{4.4}$$

$$\Delta_3(y^j, \hat{y}^j) = 0.2 \sum_i (\hat{y}^j_{i1} \neq y^j_{i1}) + 0.1 \sum_i (\hat{y}^j_{i1} \neq y^j_{i1} \& \hat{y}^j_{i2} \neq y^j_{i2}) \tag{4.5}$$

In the experiment, $\Delta_1(\cdot)$ corresponds to zero margin loss, $\Delta_2(\cdot)$ corresponds to the default margin loss function in previous experiments, and $\Delta_3(\cdot)$ corresponds to the lower margin

Experiment	Random Seed	Margin Loss functions	UAS	LAS
9	9628	$\Delta_1(\cdot)$	92.13	87.48
		$\Delta_2(\cdot)$	91.95	87.38
		$\Delta_3(\cdot)$	92.14	87.44

Table 4.10: Comparison of Margin Loss Functions

loss in the case that the head of a word is predicted correctly but their dependency type is predicted incorrectly.

The parameter settings are GloVe word embedding size 50, postag embedding size 50 randomly initialized from uniform distribution (-0.2,0.2), parameters of the neural network randomly initialized from uniform distribution (-0.01,0.01), hidden layer size 200, activation function cubic, AdaGrad learning rate 0.01, dropout rate 0.5, regularization parameter 0.0001, margin loss discount factor 0.3, batch size 20 and epoch 20. In the experiment, we keep all parameters the same except margin loss functions.

Practically, the role of the margin loss function during training is to encourage the graph search algorithm to return a high scoring tree but a rather different one than the real tree so that the score of this tree will be decreased and the score of the real tree will be increased. With a zero margin loss, the highest scoring tree will be close to the real tree, passing less information to the model as two trees cancelled each other in the training objective.

However, from Table 4.10, there is no obvious difference in terms of performance among these margin loss functions. One possible reason is that neural networks with these three margin loss functions learn equally in the beginning because parameters are pretty random and the highest scoring tree is by no means different than the real tree. As the AdaGrad learning rate is monotonically decreasing through time, even these three loss functions act differently later, those parameters which are frequently updated already become stable. In the future, we will try different adaptive gradient descent methods to investigate this potential problem caused by AdaGrad.

4.9 Analysis of Predictions

The result with highest performance we have obtained so far is from experiment 9 with $\Delta_1(\cdot)$ margin loss function. In this section, to get some insight about the source of errors made by the neural network model, we analyzed predicted trees produced by this result.

The gap between UAS and LAS is 4.64%. Given the head of a word is correctly predicted, the accuracy of predicting the dependency types is $87.48/92.12 = 94.96\%$. We would like to know what type of mistake the neural network is prone to make under this condition. Figure 4.4 and Figure 4.5 plot the confusion matrix given the head of a word is correctly predicted. Figure 4.4 plots the confusion matrix scaled by golden labels. By "scaled by golden labels", we mean the value in a confusion matrix is divided by the total number of its corresponding dependency type in a dataset. For example, suppose 9 of "NMOD" are predicted as "ADV", the total number of "NMOD" in a dataset is 10, then the value in a confusion matrix scaled by golden labels is 0.9. In Figure 4.4, on the y axis, dependency types are printed together with their total number. It shows the distribution of predicted dependency types given the total number of each dependency type in a dataset. Similarly Figure 4.5 shows the distribution of real dependency types given the total number of predictions for each dependency type.

In Figure 4.4, we see that GAP-PMOD, GAP-OBJ and GAP-NMOD are fully predicted as CONJ. However it is not a serious problem because the number of these dependency types is only one or two. Figure 4.4 and Figure 4.5, it seems that the neural network has difficulty in distinguishing between ADV and some less frequent dependency types. Checking Appendix B, we find that most of these misclassified dependency types are subclasses of ADV such as TMP, PRP, MNR, and DIR. It is therefore reasonable that the neural network fails to capture

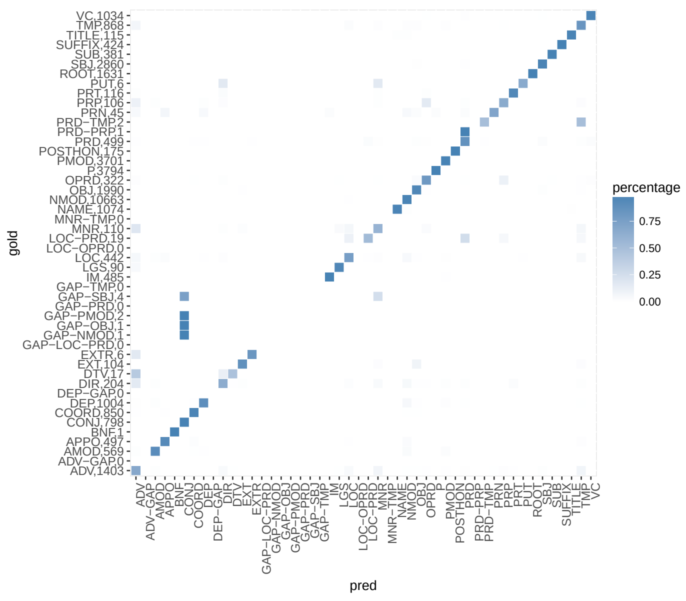


Figure 4.4: Confusion Matrix Scaled by Golden Labels

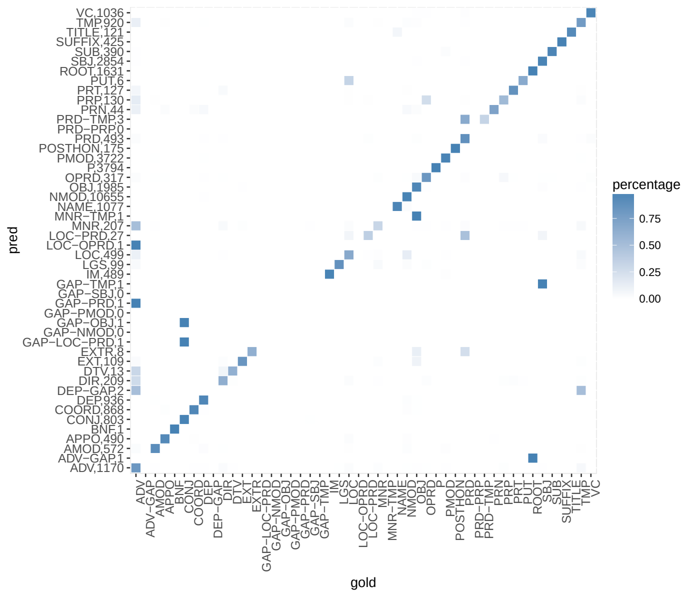


Figure 4.5: Confusion Matrix Scaled by Predicted Labels

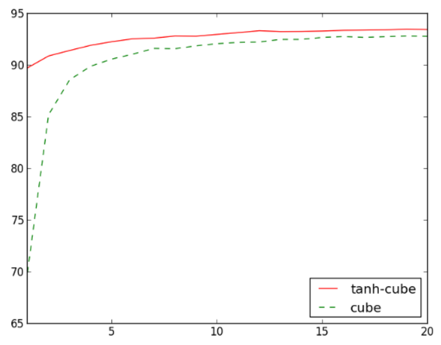
small difference between these dependency types. In general, from Figure 4.4 and Figure 4.5, as most blocks are concentrated on the diagonal line, the problem of unbalanced data mentioned in Section 4.4 has little influence on biasing model predictions.

4.10 Model Comparison

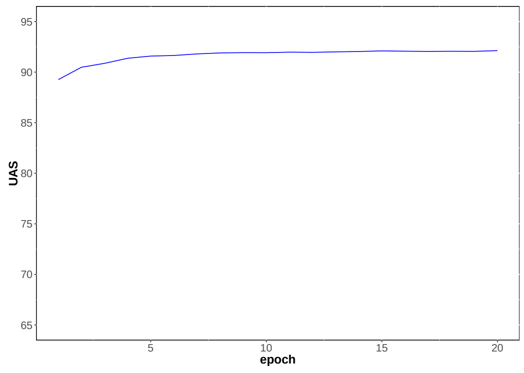
In this section we compare our model with the highest LAS obtained in Experiment 9 with Pei et al.'s model. We compare it with Pei et al.'s 1-order-atomic model. This model contains the same set of features as in our model. The 1-order-atomic refers to assuming independence between arcs and using pre-trained word embeddings. Table 4.11 reports the comparison of model performance.

	Validation		Test	
Model	UAS	LAS	UAS	LAS
Pei 1-order-atomic	92.19	90.94	92.14	90.92
This work	92.12	87.48	91.59	87.34

Table 4.11: Model Comparison



(a) Convergence plot of Pei et al.'s model



(b) Convergence plot of our model

Figure 4.6: Comparison of Convergence Plot

From Table 4.11, we reached a good UAS but a lower LAS. the difference between UAS and LAS of our model is bigger than that of Pei et al.'s model. Figure 4.6 compared convergence plots of two models. In Figure 4.6a, Pei et al. plotted the learning curve of neural networks with tanhcubic and cubic functions respectively. In Figure 4.6b, it plotted the learning curve of our neural network with the cubic activation function. As discussed in Section 4.6, the learning speed of cubic activation function in our model is faster than Pei et al.'s model.

The main difference between our model and Pei et al.'s model is that they used two neural networks to model the score of right arcs and left arcs separately while our model ignored the feature about arc directions and only used one neural network. This feature may have effects on both UAS and LAS. For example, it makes sense that the arc "drink"←"water" should be scored higher than "drink"→"water". However, as surrounding words of the arc are taken as features, the arc "drink"←"water" and "drink"→"water" can hardly have same data represention. This helps the neural network differentiate these two arcs. The lack of information about arc directions can affect the Labeled Attachment Score more because postags of two words in the arc are crucial features in determining their dependency type. For example, an left arc pointing from a verb to a noun is more likely to be labeled as OBJ than a right arc with the same postags. Pei et al.'s model has approximately 200,000 parameters more than ours. With the increase of model parameters, the computation cost is also increased. The parsing speed of Pei et al. is 55 sentence per second. However, parsing speeds of our model and their model are not comparable as the computer environment is different. In future work, we can incorporate arc directions in our model by doubling the number of neurons in the output layer. The output of a neuron in the output layer will represent the score of an arc with a specific dependency type and its arc direction. This modification only increases about 13000 parameters.

Another difference is that Pei et al. used Yamada and Matsumoto [40] head rules to convert pharsed structured Penn treebank to dependency treebank while we used head rules agreed on CoNLL 2008 [39]. Input sencetences are the same but dependency strutures are annotated differently. The number of dependency types in our dataset is more than that in Pei et al.'s dataset. This partly explains that the difference between UAS and LAS of our model is bigger than that of Pei et al.'s model.

5 Discussion

5.1 Discussion of Data

Experimental results show that the English syntactic structure is highly predictable. Patterns are evident in the training data but capturing patterns is a complex problem. One limitation of our dependency parser is that the training data is a small sample of English sentences. The training data was collected from Wall Street Journal. It covered a small portion of topics related with news. The performance of the parser needs to be examined if the test data is collected from another domain that has many unknown words to the parser. Resources of annotated data are countable while resources of unannotated data are limitless. On the one hand, it is promising to reduce the cost of annotating data with the assist of computers. On the other hand, as unlabeled data is much cheaper in natural language processing, it is worth to try semi-supervised learning [41]. A semi-supervised model uses both labeled and unlabeled data for training. Building a semi-supervised model using neural networks for dependency parsing can be a future research topic.

This thesis mainly focused on English dependency treebank. The dependency parsing system is applicable to any other languages under the condition that the training data is of the same format. In the next step we will evaluate the performance of our neural network on a Swedish dependency treebank with randomly initialized word embeddings and postag embeddings.

The raw input to a dependency parsing system is a sentence. Part-of-speech tags are assumed to be known for each word in this thesis. In real application, we need to first predict part-of-speech tags of words using a part-of-speech tagger. The accuracy of our dependency parser will be to some extent lowered down depending on the accuracy of a part-of-speech tagger. In the future work, we will use the Standford POS tagger [42], which is expected to achieve an accuracy about 97%.

The inputs of the neural network model, mainly consists of word embeddings and postag embeddings, is by itself a group of parameters that are updated in each iteration. Although we experimented initializing word embeddings and postag embeddings from a same distribution, we can not guarantee that the input data is of the same scale since it is not fixed. This could possibly prevent the neural network from finding a better local optimum before the learning rate is tuned down by AdaGrad. The batch normalization proposed by Ioffe et al.

[26] provides a solution for this problem. It normalizes the input data in each mini-batch, speeding up the learning process and making the model easier to get out of a local optimum.

Data preprocessing is crucial to the success of a project. One important step of data preprocessing in this thesis is that replacing some words in the training data as "<UNKNOWN>" so that if any words unseen in the training set occurred in the test data, it can be substituted with the embedding of "<UNKNOWN>". We simply assigned words occurred only once in the training data as unknown words. However, there is a large number of words occurred only once in the training set. This loses part of information because even though a word only occurred once in the training data it might occur again in the test data. In some sense this word is not a real unknown word. In the future work, we will randomly assign the unknown label to words in the training set inversely proportional to their frequency [6]. It will enable us to control the number of unknown words by adjusting probabilities.

5.2 Discussion of Model

The features selected in the neural network model are assumed to be core features. We did not perform feature selection in our model. Although L2 regularization is applied in the neural network model, it penalizes features on a parameter level. Features in the setting of our neural network model are mostly word embeddings and postag embeddings. This means each feature consists of 50 dimensions. Treating each embedding as a unit, there are totally 21 features in our model. Feature importance in neural network is normally evaluated by cross validation errors. In order to perform variable selection, we first need to improve the running speed of our dependency parser in the future.

In this thesis, we introduced the feedforward neural network as a generative model. The training objective of this model is to minimize the upper bound of posterior expected loss. This violates a common practice of Bayesian statistics. A Bayesian approach first makes inference about the posterior of parameters given data. Based on posteriors distribution, by Bayesian theory, one chooses an action which minimizes the posterior expected loss. However exact inference is intractable for our problem. We alternatively chose the max-margin training objective which is equivalent to the minimization of the upper bound of posterior expected loss on a training set.

Moreover, the generative neural network model can also be applied to a common classification problem. The difference is that a graph search algorithm is no longer needed. The highest score of (x, y) is simply the highest value in the output layer. One possible advantage of this generative neural network is that it deals better with unbalanced data. Because in each iteration only parameters associated with real label and predicted label are updated while in a discriminative model, all parameters are updated leading frequent labels have a bigger influence on rare labels. In this thesis, words and postags are represented by embeddings. Similarly, we can also represent categorical variables by randomly initialized embeddings rather than 0-1 representations in a common classification problem. This allows the neural network to learn the inner relationship among different categories of a categorical variable. In the future, we will further research on the performance of generative neural network models for classification problems.

The big challenge of this thesis is to implement the model by the author. A large amount of time is spent on developing a suitable neural network model for nlp task. Although an highly advanced neural network java library DL4j is available for use, they have not made it support back propagating errors to word embeddings. Besides, they do not provide the needed cost function as well as activation functions in the thesis. Their code is highly optimized making it hard to customize it according to our needs. Transferring to other languages such as python requires extra effort to rewrite part of classes in the old system. Therefore it is decided to implement the neural network dependency parser by the author. The benefits of implementing this system are that we have a better understanding of all methods being used and it is more flexible for us to try new ideas on this system.

6 Conclusion

In this thesis, the feedforward neural network model proves to be an effective model in dependency parsing. It learns intermediate feature representations and releases the heavy labor of linguistic feature engineering. We find that normalized GloVe word embedding is a better initialization than the random initialization from a uniform distribution. Second, we find the cubic activation function outperforms tangent hyperbolic and tangent hyperbolic cubic activation function. Besides, though the distribution of dependency types is highly unbalanced, the analysis of experimental results shows it does not cause the model tend to predict dependency types of the majority.

Compared to Pei et al. [5], our neural network achieves a good UAS but a lower LAS. However we can not come to an conlusion about which model is better than the other because of the use of different head rules. We intepreted the max-margin training objective as minimizing the upper bound of posterior expected loss. Under this framework, the margin loss is seen as the empirical loss. The model performance is not affected by varying the margin loss function according to actual situations. Searching for a highest scoring tree taking margin loss into account requires revising the Eisner algorithm. We proposed the Loss-Augmented Eisner algorithm to solve the problem. We introduced the feedforward neural network model as a generative model with the max-margin training objective. This generative neural network model can also be applied to a common classification problem. Rather than obtaining the highest score by Eisner algorithm, the highest score in a common classification problem is simply the maximum value in the output layer.

Bibliography

[1] Ryan McDonald, Koby Crammer, and Fernando Pereira. "Online large-margin training of dependency parsers". In: *Proceedings of the 43rd annual meeting on association for computational linguistics*. Association for Computational Linguistics. 2005, pp. 91–98.

[2] Ryan McDonald et al. "Non-projective dependency parsing using spanning tree algorithms". In: *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*. Association for Computational Linguistics. 2005, pp. 523–530.

[3] Ryan T McDonald and Fernando CN Pereira. "Online Learning of Approximate Dependency Parsing Algorithms". In: *EACL*. 2006.

[4] Terry Koo and Michael Collins. "Efficient third-order dependency parsers". In: *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics. 2010, pp. 1–11.

[5] Wenzhe Pei, Tao Ge, and Baobao Chang. "An effective neural network model for graph-based dependency parsing". In: *Proc. of ACL*. 2015.

[6] Eliyahu Kiperwasser and Yoav Goldberg. "Simple and Accurate Dependency Parsing Using Bidirectional LSTM Feature Representations". In: *arXiv preprint arXiv:1603.04351* (2016).

[7] Joakim Nivre. "Incrementality in deterministic dependency parsing". In: *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together*. Association for Computational Linguistics. 2004, pp. 50–57.

[8] Joakim Nivre. "Algorithms for deterministic incremental dependency parsing". In: *Computational Linguistics* 34.4 (2008), pp. 513–553.

[9] Daniel H Younger. "Recognition and parsing of context-free languages in time n^3". In: *Information and control* 10.2 (1967), pp. 189–208.

[10] Jason M Eisner. "Three new probabilistic models for dependency parsing: An exploration". In: *Proceedings of the 16th conference on Computational linguistics-Volume 1*. Association for Computational Linguistics. 1996, pp. 340–345.

[11] David A Smith and Noah A Smith. "Probabilistic Models of Nonprojective Dependency Trees". In: *Science Sinica*. 1965, 1396–1400.

[12] Michael Collins and Brian Roark. "Incremental parsing with the perceptron algorithm". In: *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics. 2004, p. 111.

[13] Ben Taskar et al. "Max-Margin Parsing". In: *EMNLP*. Vol. 1. 1.1. Citeseer. 2004, p. 3.

[14] Ryan McDonald and Giorgio Satta. "On the complexity of non-projective data-driven dependency parsing". In: *Proceedings of the 10th International Conference on Parsing Technologies*. Association for Computational Linguistics. 2007, pp. 121–132.

[15] Terry Koo et al. "Structured prediction models via the matrix-tree theorem". In: *Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*. 2007, pp. 141–150.

[16] Y. J. Chu and T. H. Liu. "On the Shortest Arborescence of a Directed Graph". In: *EMNLP-CoNLL*. 2007, pp. 132–140.

[17] Thomas Tutte William. *Graph Theory*. Cambrige Press, 1984.

[18] Mark A Paskin. "Grammatical digrams". In: *Advances in Neural Information Processing Systems* 14.1 (2002), pp. 91–97.

[19] Karim Lari and Steve J Young. "The estimation of stochastic context-free grammars using the inside-outside algorithm". In: *Computer speech & language* 4.1 (1990), pp. 35–56.

[20] Warren S McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.

[21] Paul Werbos. "Beyond regression: New tools for prediction and analysis in the behavioral sciences". In: (1974).

[22] David E Rumelhart and David Zipser. "Feature discovery by competitive learning". In: *Parallel distributed processing: explorations in the microstructure of cognition, vol. 1*. MIT Press. 1986, pp. 151–193.

[23] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural networks* 2.5 (1989), pp. 359–366.

[24] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization". In: *The Journal of Machine Learning Research* 12 (2011), pp. 2121–2159.

[25] Nitish Srivastava et al. "Dropout: A simple way to prevent neural networks from overfitting". In: *The Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

[26] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167* (2015).

[27] Yoshua Bengio et al. "A Neural probabilistic language models". In: (2003), pp. 1137–1155.

[28] Tomas Mikolov et al. "Distributed representations of words and phrases and their compositionality". In: *Advances in neural information processing systems*. 2013, pp. 3111–3119.

[29] Jeffrey Pennington, Richard Socher, and Christopher D Manning. "Glove: Global Vectors for Word Representation". In: *EMNLP*. Vol. 14. 2014, pp. 1532–1543.

[30] Ronan Collobert et al. "Natural language processing (almost) from scratch". In: *The Journal of Machine Learning Research* 12 (2011), pp. 2493–2537.

[31] Danqi Chen and Christopher D Manning. "A Fast and Accurate Dependency Parser using Neural Networks." In: *EMNLP*. 2014, pp. 740–750.

[32] Sandra Kübler, Ryan McDonald, and Joakim Nivre. "Dependency parsing". In: *Synthesis Lectures on Human Language Technologies* 1.1 (2009), pp. 1–127.

[33] Christopher M Bishop. "Pattern Recognition". In: *Machine Learning* (2006).

[34] Kevin P Murphy. *Machine learning: a probabilistic perspective*. 2012, p. 693.

[35] Nathan D Ratliff, J Andrew Bagnell, and Martin Zinkevich. "(Approximate) Subgradient Methods for Structured Prediction". In: *International Conference on Artificial Intelligence and Statistics*. 2007, pp. 380–387.

[36] Naum Zuselevich Shor. *Minimization methods for non-differentiable functions*. Vol. 3. Springer Science & Business Media, 2012.

[37] Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. "Building a large annotated corpus of English: The Penn Treebank". In: *Computational linguistics* 19.2 (1993), pp. 313–330.

[38] Ann Taylor, Mitchell Marcus, and Beatrice Santorini. "The Penn treebank: an overview". In: *Treebanks*. Springer, 2003, pp. 5–22.

[39] Mihai Surdeanu et al. "The CoNLL-2008 shared task on joint parsing of syntactic and semantic dependencies". In: *Proceedings of the Twelfth Conference on Computational Natural Language Learning*. Association for Computational Linguistics. 2008, pp. 159–177.

[40] Hiroyasu Yamada and Yuji Matsumoto. "Statistical dependency analysis with support vector machines". In: *Proceedings of IWPT*. Vol. 3. 2003, pp. 195–206.

[41] Eliyahu Kiperwasser, Israel Ramat-Gan, and Yoav Goldberg. "Semi-supervised Dependency Parsing using Bilexical Contextual Features from Auto-Parsed Data". In: ().

[42] Kristina Toutanova et al. "Feature-rich part-of-speech tagging with a cyclic dependency network". In: *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*. Association for Computational Linguistics. 2003, pp. 173–180.

[43] Richard Johansson. *Dependency Syntax in the CoNLL Shared Task 2008*. 2008.

A Appendix of Code Files

File Name	Description
act.java	It stores activation functions.
dataIter.java	It reads in dependency trees and produces a data matrix in each batch iteration.
DataShuffler.java	It shuffles the original treebank.
EdgeFeaturizer_NN.java	It transforms arcs into data representations.
EdgeScorer_NN.java	It scores every pair of arcs in a sentence given a neural network model.
Model_NN.java	It stores information about word embeddings, postag embeddings, dictionary of dependency types, and dictionary of postags.
Parser_NN.java	It is a replicate of Parser.java which is made compatible with the neural network model.
stats.java	It stores global methods.
Trainer_NN.java	It trains the neural network.
VocReader.java	It reads the pretrained word embeddings into memory.
cleanData.py	It preprocesses the original data.
eval.py	It evaluates the performance of dependency parser excluding punctuations.
normalData.py	It standardizes the GloVe word embeddings.

Table A.1: the Description of created Files

File Name	Description
CoNLLReader.java	It reads in dependency trees from a text file.
CoNLLTree.java	It creates the data structure for a dependency tree.
CoNLLWriter.java	It writes predicted dependency trees into a text file.
Parser_NN.java	It performs Eisner algorithm.
Table.java	It creates the data structure for the dictionary of dependency types and the dictionary of postags.

Table A.2: the Description of existed Files

B Appendix of Dependency Labels

Table B.1 provides description of dependency labels provided by Surdeanu et al. [39], Johansson and Richard [43]. Labels might be combined such as ADV-GSP.

Label	Description
ADV	General adverbial
AMOD	Modifier of adjective or adverbial
APPO	Apposition
BNF	Benefactor complement (for) in dative shift
CONJ	Second conjunct (dependent on conjunction)
COORD	Coordination
DEP	Unclassified relation
DIR	Adverbial of direction
DTV	Dative complement (to) in dative shift
EXT	Adverbial of extent
EXTR	Extraposed element in expletive constructions
GAP	Gapping: between conjunction and the parts of a structure with an ellipsed head
HMOD	Modifier in hyphenation, such as two in two-part
HYPH	Between first part of hyphenation and hyphen
IM	Infinitive verb (dependent on infinitive marker to)
LGS	Logical subject
LOC	Locative adverbial or nominal modifier
MNR	Adverbial of manner
NAME	Name-internal link
NMOD	Modifier of nominal
OBJ	Direct or indirect object or clause complement
OPRD	Object complement
P	Punctuation
PMOD	Between preposition and its child in a PP
POSTHON	Posthonorific modifier of nominal such as Jr, Inc.
PRD	Predicative complement
PRN	Parenthetical

Label	Description
PRP	Adverbial of purpose or reason
PRT	Particle
PUT	Various locative complements of the verb put
ROOT	Root
SBJ	Subject
SUB	Subordinated clause (dependent on subordinating conjuction)
SUFFIX	Possessive suffix (dependent on possessor)
TITLE	Titles such as Mr, Dr
TMP	Temporal adverbial or nominal modifier
VC	Verb chain
VOC	Vocative

Table B.1: Description of Dependency Types