Linköping University | Department of Computer and Information Science
Bachelor thesis, 16 credits | Computer programming
Spring term 2017 | LIU-IDA/LITH-EX-G--17/011—SE

Transition-Based Dependency Parsing with Neural Networks

Joakim Gylling

Supervisor, Rita Kovordányi
Co-supervisor, Marco Kuhlmann
Examiner, Peter Dalenius

ABSTRACT

Dependency parsing is important in contemporary speech and language processing systems. Current dependency parsers typically use the multi-class perceptron machine learning component, which classifies based on millions of sparse indicator features, making developing and maintaining these systems expensive and error-prone. This thesis aims to explore whether replacing the multi-class perceptron component with an artificial neural network component can alleviate this problem without hurting performance, in terms of accuracy and efficiency. A simple transition-based dependency parser using the artificial neural network (ANN) as the classifier is written in Python3 and the same program with the classifier replaced by a multi-class perceptron component is used as a baseline. The results show that the ANN dependency parser provides slightly better unlabeled attachment score with only the most basic atomic features, eliminating the need for complex feature engineering. However, it is about three times slower and the training time required for the ANN is significantly longer.

INTRODUCTION

Natural Language Processing (NLP) is an area of research that explores how computer systems can be created and used to understand and manipulate natural language text or speech in a useful manner. It is the area of research behind Google Translate, spellchecking programs and many more applications we often use in our everyday lives.

NLP faces several challenges, one of which is ambiguity. Most words have different meaning in different context. It is easy for us humans to figure out the intention of the words by looking at the context they appear in, but how can this be achieved for a machine? The idea of "context" is very fuzzy and situational, thus it cannot simply be hardcoded. One way to help figure out the meaning of the words in a sentence is to create a dependency graph with part-of-speech tagging. Figure 1 illustrates how a dependency graph typically looks like.

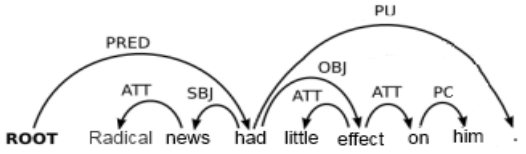

Figure 1: An illustration of a dependency graph.

Dependency parsing is the task of mapping a natural language sentence into a representation of its syntax or semantics in the form of a dependency graph. Dependency graphs can be created in linear time complexity with transition-based dependency parsing [4].

In standard experiments using the Penn Treebank [14] as the training and test data source, labeled and unlabeled attachment scores of over 90% have previously been achieved by transition-based dependency parsers using linearly separable machine learning algorithms such as the multi-class perceptron [4, 5]. However, many of these rely on millions of hand-crafted features, making developing and maintaining these systems expensive and error-prone. Models based on artificial neural networks [1] as the machine learning component promise a solution to this problem, as they are able to learn new features automatically.

Purpose

The goal of this project is to get a better understanding of the possibilities and limitations of neural networks in the context of a system for transition-based dependency parsing and to provide an implementation of such a system.

Focus will be placed on simplicity rather than cutting-edge performance, with the intention to construct a system that is easy to apply small specific changes to for experimentation purposes.

Honnibal [3] has developed a compact system for transition-based dependency parsing based on the multi-class perceptron. A similar implementation will be provided, were the current machine learning model is replaced by an appropriate neural network model.

Research question

Given the system provided by Honnibal [3]:

- *What are the pros and cons of using a neural network instead of the multi-class perceptron as the machine learning component?*
- *How high is the unlabeled attachment score for the new system, using standard test from the English Universal Dependency Treebank [24] and how well does that fare in comparison with Honnibal's original system?*

Delimitations

Several transition-based parsers with neural networks have been explored and produced very promising results in recent times [8, 9, 10, 11, 12, 13, 17]. The main goal of these articles is mostly about finding a better and more accurate algorithm than the, at the time, current state of the art. The focus of this project, however, is to compare a simple implementation with one specific modification and analyze the experimental findings.

The created system is written in Python3.6.0. The state of the art neural network library, Keras [29], is used to simplify the neural network implementation process significantly. The Python library gensim [31] is used to create Word2Vec models in this thesis. Word2Vec [30] is a program that maps words to fixed size vector representations based on semantic information.

THEORY

There are a lot of techniques and concepts involved in the context of dependency parsing and machine learning algorithms, and a lot of them are very complex. This section will attempt to give a high-level overview of the most relevant techniques and concepts.

Dependency parsing

The computerized task of mapping a natural language sentence into a dependency graph is a classification problem at its core. The algorithm finds correlated pairs of tokens in a natural language sentence and classifies each such pair with its specific syntax type. Every pair is basically a directed arc consisting of one "head" pointing to its "dependent". Furthermore, every standalone token will also oftentimes be classified with its own syntactic type.

A dependency graph is a directed graph where every standalone token is a vertex, and every mapped pair of tokens is an arc [21]. The graph satisfies the following constraints:

- There is a single root vertex that has no incoming arcs.
- With the exception of the root vertex, each vertex has exactly one incoming arc.
- There is a unique path from the root vertex to each vertex in the graph.

As shown by Figure 1, it is common practice in dependency parsing to add a dummy node marked as ROOT to facilitate connectedness to the graph.

Dependency parsing is a problem that requires some kind of search strategy that can either be categorized as "greedy", "exhaustive", or a hybrid of the two. A greedy search strategy relies on local information when making the classification decisions, and an exhaustive search relies on global information. It is essentially these two search strategies that further divide a dependency parsing strategy into either a transition-based or a graph-based dependency parsing strategy.

Transition-based dependency parsing

The process of a transition-based dependency parser can generally be explained as a procedure that follows these steps:

1. The parser starts in the initial configuration.
2. It uses a classifier to predict the transition that should be made to move to the next configuration.
3. It repeats step 2 until it reaches a terminal configuration.

The classifier uses a greedy search algorithm on local information accessible from the current configuration to determine what transition to be made next. Inference by a trained multi-class perceptron component is the most commonly used greedy search strategy for transition-based dependency parsing.

Because the search space is limited to the current configuration at each step, the runtime complexity of this approach is linear in the number of transitions and hence in the length of the sentence. A major disadvantage, on the other hand, is that the greedy parsing strategy may lead to error propagation. Another disadvantage is that the transition-based approach can only produce projective dependency graphs, but not all sentences are projective, as shown by Figure 2. So there will necessarily be some errors in such sentences [21].

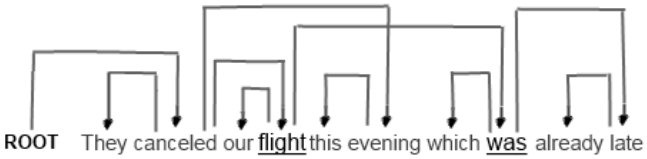

Figure 2: An example of a non-projective dependency graph. A graph is projective if all arcs are projective. An arc from head to dependent is projective if there is a path from the head to all the tokens in between the arc. This is not the case for the arc between the head 'flight' and the dependent 'was' in this example.

The transitioning during the parsing can be done in several different ways. The three most commonly used algorithms for transitioning are the arc-standard, the arc-eager and the arc-hybrid, as explained in great detail by Nivre [23]. The arc-standard is a simple variant with only three possible transition actions, whilst the arc-eager is slightly more complex with four possible transitions and 3 constraints. Arc-hybrid is very similar to arc-standard but with a few additional constraints.

Graph-based dependency parsing

Graph-based parsers use an exhaustive search on global data from the whole space of possible trees to find an optimal solution for the current problem. It is generally more accurate than transition-based dependency parsing but suffers from efficiency problems. There are algorithms that make it possible to limit the time complexity to polynomial time of the sentence length, such as the CKY (Cocke-Kasami-Younger) algorithm, so it is definitely a worthy approach when accuracy is of importance.

Zhang and Zhao [7] combined a convolutional neural network in combination with a graph-based dependency parsing and yielded good results, both in regards to runtime and accuracy.

Hybrid

Combining the two approaches for dependency parsing and only receiving the best from two worlds is an obvious idea. It is fairly difficult to accomplish since they rely on two completely different scopes of data, but it is possible and can yield good results, as shown by Zhang, Nivre and McDonald [25, 22].

Machine learning algorithms

Every sentence consists of words, and every word consists of letters. The parts are not the same as the whole in this

case, because the whole carries meaning that cannot be found in the parts. For a computer to "understand" the meaning of a sentence, a representation of the syntactic structure, with relevant metadata like word relations etc., is a requirement. A dependency graph is a sufficient syntactic representation for this. It is, however, not an easy task to create a handcrafted function that can parse an arbitrary sentence into its correct dependency graph. For an algorithm to be able parse it into its syntactic structure it would possibly need a database of all possible sentences and their correct syntactic structure, which is not feasible. For that reason a machine learning algorithm is needed.

Machine learning is a kind of algorithm that uses data to learn how to approximate functionalities rather than being explicitly coded for that functionality from the start [26]. Sometimes it is too difficult for a certain functionality to be understood and implemented efficiently and that is when machine learning algorithms come to the rescue.

In the context of dependency parsing the most used machine learning algorithmic approach is the multi-class perceptron. In more recent times neural networks have made great progress and become popular [8, 9, 10, 11, 12, 13, 17].

Perceptron

A perceptron essentially consists of three parts.

- A vector of input variables commonly referred to as features.
- A set of output variables.
- A set of weights.

Every feature have as many weights as there are outputs, and every one of these weights is connected to one of the outputs, as shown by Figure 3.

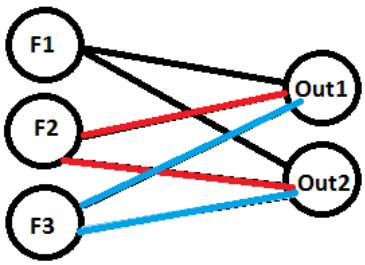

Figure 3: An example of a perceptron algorithm with 3 features (F1, F2, F3), their corresponding weights (drawn as lines) and 2 outputs (Out1 and Out2).

The perceptron is trained from provided data via supervised learning [26]. Training involves making changes to the weight values. By extracting features from a sentence a calculation will occur and the result will end up in the output. Depending on the values of the outputs the system will determine what class it represents.

Multi-class perceptron

The multi-class perceptron is an extension of the standard perceptron introduced by Collins [15]. The standard perceptron has a binary classification, meaning that every input will be predicted to either belong to class A or class B, no other choices are available. The multi-class perceptron, on the other hand, can have a huge amount of possible classes available and it is not simply a single prediction involved in the larger context. It is sequence of predictions that include previous class predictions as features when making future predictions and the final result is the whole sequence of the predicted classes.

In the context of transition-based dependency parsing the predicted class at each step is the correct transition action given the current configuration. The predicted action will be taken and give rise to a new configuration with new available local information extracted as features to make yet another prediction and this process is repeated until terminal configuration is reached. The final result is one of all possible dependency graphs, for the given sentence.

In practice the dependency parsing will extract millions of features based on, for example, specific word combinations and place them with their corresponding weights in a table during the training phase. After that, during the prediction phase, a set amount of information will be extracted from the current configuration and looked up in said table. This extraction step is expensive and causes most of the runtime to be consumed in this step [27], in some cases.

Artificial neural network

Artificial neural network is pretty much like an extended and more advanced version of a multi-class perceptron. The basic idea with weights, inputs (features) and outputs are the same, but there is one or more extra layers of vectors added in neural networks (called hidden layers) and the learning algorithm is significantly different. Another difference is that neural networks use an activation function, which a multi-class perceptron does not. For the explanation and details regarding all of these concepts I will refer to Demuth's book [1].

Why artificial neural networks instead of multi-class perceptron?

The important point is that the multi-class perceptron suffer from the disadvantage that they are only able to solve linearly separable classification problems. To cope with that, a huge amount of hand-crafted features is required. Artificial neural network, on the other hand, is able to approximate almost any function, if it has a sufficient amount of neurons in its hidden layers, and can solve non-linear separable problems as a result [1]. This makes it possible for neural networks to accurately make predictions when presented by only the most basic atomic features used by the multi-class perceptron and still come out with better accuracy. In fact, this has already been demonstrated by Chen and Manning [11].

Although inefficiency is a potential problem with the complex feature extractions required by the multi-class perceptron [27], it is not the biggest problem. The biggest problem is that the hand-crafted features require a good deal of tweaking and expert knowledge to perform satisfactory, and even then it is often biased towards the current language and test data. The same features might need to be altered once again when presented with a different set of data. A model using a neural network model does not suffer from this problem because the features used are just basic atomic information. The hidden layers of the network are responsible for finding the relevant patterns from these so the need for manual tweaking disappears. Just feed the network with a sufficient amount of good training data and it will "engineer itself", so to speak.

Dense data representation

A typical data representation for the multi-class perceptron is to extract a huge amount of different word combinations that appear in the sentences from the training set as features, and this will make the data representation extremely sparse. This representation will often contain millions of such word combinations, but only a few of them will be relevant at any given time when parsing a sentence. It is an unnecessary expense of the runtime.

The multi-class perceptron requires such sparse representations due to the limitations of the algorithm, but this is not the case for a neural network. By using the computer program Word2Vec the available atomic information can be represented as a dense vector instead and this has been shown to work very well when parsing with a neural network component [11].

Word2Vec

Word2Vec is a program that creates dense vector representations of words based on semantic information [30]. The semantic information is based on the idea that similar words occur in similar contexts. Word2Vec implements two methods to create said vector representations.

1. Continuous skip-gram: Predicting a context based on a word.
2. Continuous bag of words (CBOW): Predicting a word based on the context.

There exist several excellent pre-trained Word2Vec models freely available on the Internet. The most popular at the time of writing is the Google news dataset model[1]. It has been trained on roughly a 100 billion words and has a vector length of 300 features.

The Python library gensim [32] is used in this thesis to produce a Word2Vec model for the parser.

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

Part-of-speech tags

The information available as features when dependency parsing includes part-of-speech tags (POS-tags). The process of automatically predicting POS-tags is called part-of-speech tagging (POS-tagging). It is a classification problem that generally requires inference from a machine learning component as its search strategy, similarly to dependency parsing. The multi-class perceptron is the most commonly used machine learning component for POS-tagging, but artificial neural networks can also be used with good results, as shown by Strandqvist [6].

Honnibal's original system [3]

The following summarizes Honnibal's implementation.

- It is written in Python2.7 with approximately 500 lines of code.
- It is a transition-based dependency parser using the arc-hybrid transition algorithm [23].
- It uses the algorithm "dynamic oracle" [28] to predict gold-standard moves at runtime as a way to recover from errors and avoid error propagation.
- It raises an error when presented with non-projective dependency graphs as training data.
- The parser only predicts unlabeled dependency information and POS-tags.
- It uses the multi-class perceptron as the machine learning component.
- It has its own POS tagger, which also uses the multi-class perceptron as the machine learning component.
- The POS-tagger has a dictionary of common words with almost constant tags independent of the context. These words are located in the training phase and helps speed up the parsing time.

Honnibal's system was rewritten in Python3.6.0 and used as the baseline program for this thesis. It was tested on the same machine with the same environment as the newly created system.

Evaluation

When measuring dependency parsing accuracy, there are three common strategies to use. They are the following:

- Labeled Attachment Score (LAS): Percentage of tokens for which the system has predicted the correct head and dependency relation.
- Unlabeled Attachment Score (UAS): Percentage of tokens for which the system has predicted the correct head.
- Label Accuracy (LA): Percentage of tokens for which the system has predicted the correct dependency relation.

Unlabeled attachment score is the only measurement used in this thesis, due to the limitations of the baseline system.

Output layer:

$$p = softmax(W_2 h)$$

Hidden layer:

$$h = sigmoid(W_1^w x^w + W_1^t x^t + bias)$$

Input layer:

$$[x^w, \quad x^t]$$

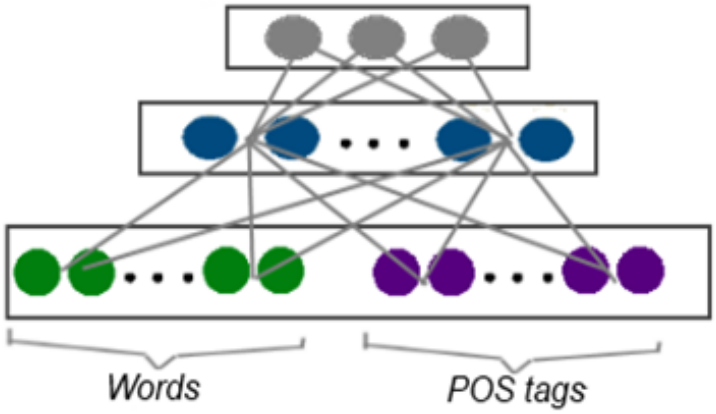

Figure 4: My Neural network architecture. The concatenated word vectors are denoted as x^w and the tag vector is denoted as x^t. The lines between the hidden layer and the input layer represent the weight vector denoted as W_1 and the lines between the output layer and hidden layer represent the weight vector denoted as W_2

Treebanks

Modern dependency parsers use data from corpora of syntactic analyses called Treebanks to train their machine learning components. There are many such Treebanks available, but the most commonly used one, in the context of dependency parsing, is The Penn Treebank [14, 16].

The Penn Treebank is a huge corpus, consisting of approximately 3 million words of American English, with part-of-speech tagging and skeletal syntactic structure.

There exist a collection of smaller freely available corpora compiled and released by the Universal Dependencies project (UD) [24]. The English UD data is the one mostly used as training and test data in this thesis. It contains 229,753 tokens and 14,545 sentences.

A more recent treebank that expanded upon the Penn Treebank, but with more information included enabling significantly better automatic semantic analysis, is the OntoNotes project [19, 20]. Goldberg and Orwant have also produced a huge corpus even more recently based on the English Google books corpus, consisting of text from 3,473,595 English books [2].

These are all corpora based on English texts, but there exists large and sophisticated corpora based on other languages as well. For example: The Prague Dependency Treebank [18] contains ~1.5 million Czech word tokens.

Gold-standard for transition-based dependency parsing

One of the core ideas of transition-based dependency parsing is to predict a sequence of transition actions via a supervised machine learning algorithm and that means that it needs training data with provided desired output. The treebanks is a collection of such data. Usually they are created manually by experts and this reliable data is commonly referred to as gold-standard. However, the gold-standard for transition-based dependency parsing is a sequence of transition actions which is not originally available from given training data found in a Treebank. For that reason a preprocessing of the training data where the action sequence for the given dependency graphs needs to be computed. An efficient algorithm to compute this at runtime is the "dynamic oracle" [28].

METHOD

In essence, this thesis aims to answer how and if a simple neural network can improve performance in the context of transition-based dependency parsing, and evaluate potential problems with its implementation. In order to reach a well-defined conclusion, Honnibal's system [3] was used as a baseline for the implementation and for the measurement of the efficiency and complexity.

This chapter will explain the neural network architecture, the experimentation process, the implementation and provide pseudocode for the parsing.

The computer device used for testing was a Hewlett-Packard; model p6770sc. It had 8.0 GB RAM and the operative system was a 64-bit Windows 7. The processor was a 2.80GHz Intel Core i5-2300.

Artificial neural network architecture

Figure 4 illustrates the neural network design chosen for this thesis. The same design is used for both the POS-tagger and the dependency parser. The only difference between the two is the amount of neurons in each layer. Table 1 provides the details.

	POS-tagger	Dependency parser
Output layer	53	3
Hidden layer	100	80
Input layer	2689	5825

Table 1: Number of neurons in the neural network models.

The training algorithm for the ANN is the mini-batch gradient descent. Gradient descent is a way to minimize an objective function $J(\theta)$ with parameters $\theta \epsilon \mathbb{R}^d$ by updating the parameters in the opposite direction of the gradient to the objective function $\Delta_\theta J(\theta)$. The equation is shown below (Eq1)

$$\theta = \theta - \eta \cdot \Delta_\theta J\left(\theta; x^{(i:i+n)}; y^{(i:i+n)}\right) \qquad \text{Eq1}$$

η is the learning rate constant, n is the size of the mini-batch, $x^{(i)}$ is the training example and $y^{(i)}$ is the desired output.

Experimentation process

The programs were executed via the terminal with the training and test data filenames as arguments. The time library of python was used to measure the time taken for the programs at different breakpoints. The parsing time refers to the time taken to parse all sentences from the test set (size ~2000 sentences) and the training time refers to the time taken to train on all sentences from the training set (size ~12000). The parsing time was further divided into 3 steps for the dependency parsers: feature extraction step, prediction step, and POS-tagging step.

The unlabeled attachment score was calculated by parsing the test set after each training iteration and comparing the results with the desired output.

Implementation

Honnibal's implementation [3], available on GitHub, was rewritten in Python3.6.0. A few modifications regarding the reading of training- and test-data were required, and a function that removes all non-projective graphs from the training set has been implemented. These were the only modifications applied for the baseline program.

The required modifications were necessary for the following reasons:

1. The original system throws an error when non-projective graphs are encountered.
2. The original system was created with the format of the Penn Treebank in mind, but the format of the English Universal Dependency files differs slightly. Both have the same structure with word information in each line separated with tabs, but the order and amount of available information is different.

The system mainly consists of two parts; a part of speech tagger and a dependency parser. Both use the multi-class perceptron as the machine learning component in the baseline system. The new system effectively replaced both the POS taggers perceptron and the parsers perceptron with an appropriate feedforward ANN model without changing the overall functionality.

POS Tagger

The final ANN created for the POS tagger is a basic feedforward model that is very similar to Strandqvist's [6] implementation with the following hyper parameters:

1. Learning rate: 0.1.
2. Activation function: Sigmoid for the hidden layers and softmax for the output layer.
3. Batch size: Sentence length.
4. Hidden layers: 1.
5. Hidden units in the hidden layer: 100.
6. Features: Word context set to 3 and the 3 previous tags represented as sparse vectors.

The context refers the number of words prior to and after the current word in the given sentence that will be encoded with Word2Vec and concatenated to the input vector as features for the network. Figure 5 illustrates a concrete example of a configuration and the chosen features in it.

			Current				
Words	From	the	AP	comes	this	story	:
Tags	IN	DT	NNP	VBZ	DT	NN	:
Context features	<start>	From	the	AP	comes	this	story
Tag features	<start>	IN	DT				

Figure 5: A short sentence from the training dataset and its chosen features when context is set to 3. The current word in this example is "AP".

Every word feature was encoded into a dense vector of size 300 with the Word2Vec model. The Word2Vec model used in the implementation is the popular "Google news dataset" model previously mentioned. It is a very reliable model with lots of good semantic information for most occurring words. It takes roughly 1 minute to load the model, and the lookup speed at runtime is fast enough to be negligible. One problem with it, however, is that there are several tokens that don't exist in the vocabulary of this Word2Vec model. The loaded model is an untrainable "keyedVector" so the missing words cannot be decoded and added manually to it. If all of these unknown words are represented as a single extra binary value the performance of the tagger becomes unsatisfactory, and if they are added as a sparse "one-hot" vector the size of the network becomes way to large and inefficient. To solve this issue an "UNK dictionary" with applied normalization on the missing tokens was created and added to the word vector representation as a small "one-hot" sparse vector. Only common unknown words from the training dataset, with a threshold set to at least 5 occurrences, has a place in this dictionary and with the current training dataset the vector representation for each word increased from 300 to 360.

There is of course a possibility of using the Word2Vec algorithm to train my own model that does include the missing tokens without the need for said "UNK dictionary". However, a huge amount of training data would be required for that and possibly several days of training time, which was something I didn't have.

The tags were represented as a "one-hot" sparse vector with the length being the number of all uniquely occurring tags found in the training dataset. With the English Universal Dependency training dataset the length of the vector became 53.

The total length of the input vector for the implemented POS-taggers artificial neural network was 2689, given the English Universal Dependency training set and with a context size of 3. To put this in perspective, the original

multi-class perceptron POS-tagger had a sparse "one-hot" input vector of a size larger than 70,000.

The other hyper parameters were mostly adjusted to the current choice through trial and error.

Dependency parser

The final ANN created for the dependency parser was a basic feedforward model almost identical to the ANN for the POS tagger. The following hyperparameters were used.

1. Learning rate: 0.2.
2. Activation function: Sigmoid for the hidden layers and softmax for the output layer.
3. Batch size: Sentence length.
4. Hidden layers: 1.
5. Hidden units in the hidden layer: 80.
6. Features: 14 words from the current configuration and their corresponding POS tags. See table 2 for the details.

Word features	Total (14)	Abbreviation
Top 3 words from the buffer	3	b0,b1,b2
Top 3 words from the stack	3	s0,s1,s2
2 leftmost children of s0 and b0	4	
2 rightmost children of s0 and b0	4	
Tag features	**Total (14)**	
The corresponding POS tags of all word features		

Table 2: The feature template for the ANN parser.

Every word feature was encoded into a dense vector of size 300 with the Word2Vec model, plus 60 from the concatenation of the "UNK dictionary", exactly the same as the POS tagger. The atomic features were 14 words from the current configuration and their corresponding POS tags. The size of the input vector for the implemented dependency parser was 5825 in total, given the English Universal Dependency training set. To put this in perspective, the original multi-class perceptron dependency parser had a sparse "one-hot" input vector of a size larger than 1,000,000.

Pseudocode of the parser

A simplified (incomplete) pseudocode of the implementations parsing process, given a sentence as a list of words:

1. Predict the corresponding POS-tags for the words in the given sentence with a POS-tagger. *Returns a list of POS-tags.*
2. current_word ← 0 (integer, also points to the corresponding POS-tag)
3. list_of_heads ← empty list of tuples
4. While not terminal(current_word)
   a. Extract features from the current configuration. *Returns a fixed size list of floats (the input vector).*
   b. Make prediction based on the extracted features. *Returns an action.*
   c. Take action and update current_word and list_of_heads accordingly.
5. Return the list_of_heads.

The above pseudocode was the same for both the baseline parser and the ANN parser. The only differences were at step 1, 4.a and 4.b. The differences were: different models used when predicting in step 1 and 4.b, and different feature representations returned at step 4.a.

RESULTS

The objective experimental findings of using a neural network for the POS tagger and dependency parser are presented in this chapter.

POS tagger

The original multi-class perceptron POS tagger is more efficient than the ANN tagger, but the ANN has slightly better accuracy.

The average time taken to parse 23625 tokens with the multi-class perceptron model is ~1.8 seconds and the average time for the ANN model with context size set to 3 is ~3.0 seconds (see Table 3).

	ANN	Perceptron
Training time	80 sec/iter	10 sec/iter
Tagging time	3.0 seconds	1.8 seconds

Table 3: Training and tagging time taken for the ANN tagger and the baseline tagger.

The accuracy of the multiclass perceptron model usually settles around 93.3% accuracy and the ANN model usually settles at 93.5% accuracy, when trained on the English Universal Dependency training dataset and tested on the corresponding test set. This is illustrated in figure 6 and 7.

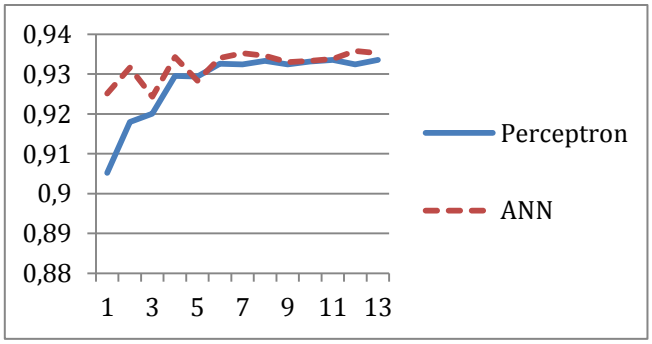

Figure 6: A typical training session of the ANN and the original multi-class perceptron implementation for POS-tagging using the same English Universal Dependency datasets. The X-axis represents number of iterations and the Y-axis represents unlabeled attachment score in decimal form.

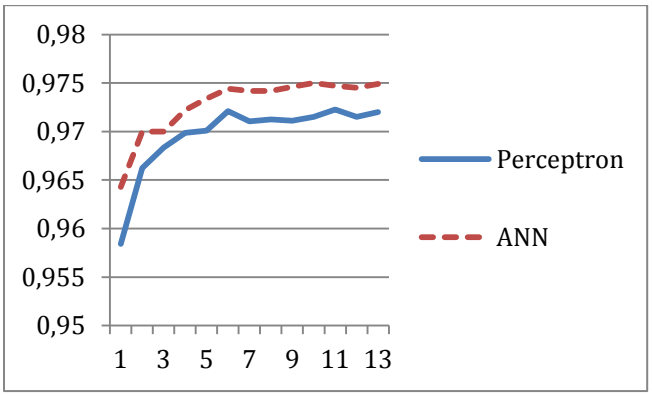

Figure 7: The same Perceptron and ANN model tested and trained on another training- and test-dataset downloaded from the website cnts[2].

Dependency parser

The ANN parser achieved an accuracy of approximately 1.5% higher than the baseline program, but the parsing speed is around 3 times slower.

Table 4 illustrates the parsing time and training time for the two parsers. Table 5 illustrates the parsing time in greater detail.

	ANN	Perceptron
Training time:	6 min/iter	1 min/iter
Parsing time:	22 seconds	7 seconds

Table 4: Parsing and training time taken for the ANN parser and the baseline parser.

Detailed parsing time		
	ANN	**Perceptron**
Feature extraction	7.9 seconds	2.7 seconds
Prediction	10.2 seconds	2.8 seconds
Tagging	3.0 seconds	1.8 seconds

Table 5: parsing time in more detail. The above three steps are the most time consuming when parsing.

The accuracy for the ANN parser is at best slightly above 82.0% and for the multi-class perceptron it is slightly below 80.7% on the English Universal dependency datasets. This is illustrated in figure 8.

[2] http://www.cnts.ua.ac.be/conll2000/chunking/

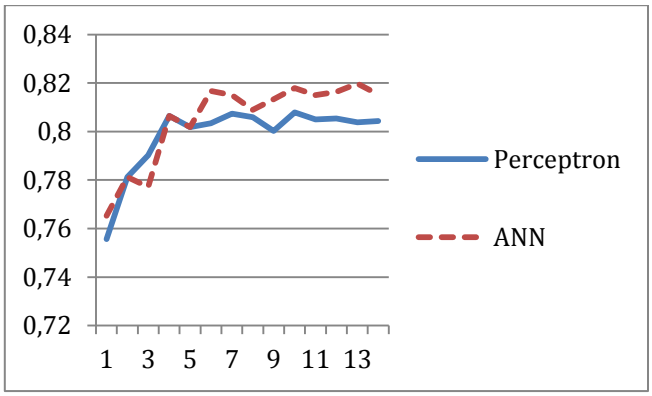

Figure 8: A typical training session of the ANN and the original multi-class perceptron implementation for dependency parsing using the same English Universal Dependency datasets. The X-axis represents number of iterations and the Y-axis represents unlabeled attachment score in decimal form.

Because simplicity is of relevance in this thesis, a small experiment to see how the training size affected the performance and training time was carried out. The parsing time was seemingly unaffected, but the training time was affected linearly with the training size, as one might expect. The results are illustrated in figure 9.

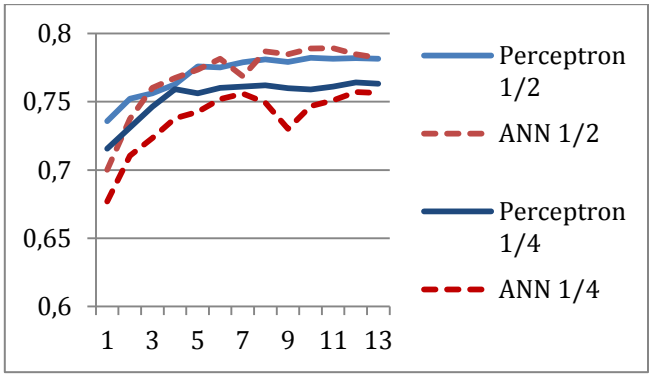

Figure 9: The same Perceptron and ANN dependency parsers trained on half the English Universal Dependency training data (the top two lines) and one fourth of the training data (the two bottom lines). They were tested on the same unmodified English Universal Dependency test-dataset.

DISCUSSION

A subjective analysis of the experimental findings is presented in this chapter.

Complexity

The aim of this thesis is to create a good but simple dependency parser, including its POS tagger. If complexity is measured in the number of lines of code, then both parsers have almost the same complexity. Both of them consist of approximately 600 lines of code which is considered very small in the context of dependency parsing. It is, however, not that simple to judge the complexity of

the program. The ANN parser relies heavily on external libraries such as Keras and Gensim, which cannot be regarded as simple. If I were to write my own functionality imitating the functionality used from said libraries the amount of code would easily exceed 1000 extra lines of code.

Another thing to consider when judging the complexity is the amount of required knowledge for the 2 implementations. ANN is a very broad machine learning algorithm with a huge amount of different variations that may or may not work well at solving the given problem. As an example, I personally tested five different activation functions for the hidden layer before settling on the sigmoid function, which produced best results on this particular implementation. The multiclass perceptron requires complex feature engineering to perform well, but aside from that it is very straightforward and relatively simple with few possible alterations.

Another problem with the ANN is that it is difficult to know what architecture works best for the problem, and the most common way to figure it out is to use trial and error. This would not be a problem if it weren't for the fact that the training time is very slow. In my case I had to wait around 30 minutes for every new tweak to be able to determine whether the change was an improvement or not. An attempt to decrease the size of the training dataset to speed up the process were made, but it turns out that even though the training time is halved if the training dataset is halved, the performance of the ANN is significantly reduced at a much faster rate than the multi-class perceptron. Whether it would still be possible to determine if a tweak is good or bad from this setup is not certain at this point, but it is a possibility. I will leave the conclusions regarding that to future work.

It does indeed seem like the ANN implementation is more complex and time consuming to implement properly than the multi-class perceptron, but there is one thing it does much better. It does not require any particular feature engineering. All it requires are a few atomic features from the locally available information. Combinations of many of the atomic features are essential for the multi-class perceptron to be able to perform satisfactory, and depending on the language the best combinations will vary. The experiments of this thesis showed that an ANN model can indeed capture enough information to perform well without the need of any such feature engineering.

Performance

The ANN models perform better than the baseline models in both the POS tagger and the dependency parser, in terms of accuracy (unlabeled attachment score). However, the difference is quite small. Both perform pretty bad compared to modern parsers trained and tested on the Penn Treebank datasets and one reason could be difference in size of the training data. Another reason could be the lack of additional information from the available configuration during the dependency parsing process in the form of dependency labels. Because both the baseline model and the ANN models in this thesis were created with simplicity in mind the dependency labels have been completely left out.

Efficiency

Previous works [11, 27] mentioned that the feature extraction process and the sparse vector representation is very time consuming for typical multi-class perceptron based dependency parsers, but this does not seem to be the case with the baseline system. The ANN parser created in this thesis extracts less information from the configuration but still takes twice as long time in this step than the baseline parser. The reason seems to be the building of the dense input vector at runtime. The perceptron parser only need to lookup the relatively few extracted features in the weight table and do a calculation with their corresponding weights and all the other zeros are ignored in the calculation. The ANN needs to create a dense representation of every feature and calculate all the values in the created dense input vector in two steps and this is shown to be even more time consuming.

CONCLUSION

This thesis has shown that a transition-based dependency parser using a simple artificial neural network with a purely atomic feature representation can indeed slightly outperform an identical system that uses the multi-class perceptron in terms of accuracy, eliminating the need for complex feature engineering. However, it is about three times slower and the training time required for the ANN dependency parser is significantly longer.

REFERENCES

1. DEMUTH, Howard B., et al. *Neural network design*. Martin Hagan, 2014.
2. GOLDBERG, Yoav; ORWANT, Jon. A dataset of syntactic-ngrams over time from a very large corpus of english books. In: *Second Joint Conference on Lexical and Computational Semantics (* SEM)*. 2013. p. 241-247.
3. HONNIBAL, Matthew. Parsing English in 500 Lines of Python. In explosion.ai [interactive]. 2013. [Previewed 2017- 02-01]. Access through internet: <https://explosion.ai/blog/parsing-english-in-python>.
4. NIVRE, Joakim. *Inductive dependency parsing*. Springer Netherlands, 2006.
5. ZHANG, Yue; NIVRE, Joakim. Transition-based dependency parsing with rich non-local features. In: *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: short papers-Volume 2*. Association for Computational Linguistics, 2011. p. 188-193.
6. STRANDQVIST, Wiktor. Neural Networks for Part-of-Speech Tagging. 2016.

7. ZHANG, Zhisong; ZHAO, Hai; QIN, Lianhui. Probabilistic graph-based dependency parsing with convolutional neural network. In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*. 2016. p. 1382-1392.
8. VASWANI, Ashish; SAGAE, Kenji. Efficient structured inference for transition-based parsing with neural networks and error states. *Transactions of the Association for Computational Linguistics*, 2016, 4: 183-196.
9. ALI, Basirat; NIVRE, Joakim. Greedy Universal Dependency Parsing with Right Singular Word Vectors. In: *Swedish Language Technology Conference (SLTC)*. 2016.
10. WEISS, David, et al. Structured training for neural network transition-based parsing. *arXiv preprint arXiv:1506.06158*, 2015.
11. CHEN, Danqi; MANNING, Christopher D. A Fast and Accurate Dependency Parser using Neural Networks. In: *EMNLP*. 2014. p. 740-750.
12. DYER, Chris, et al. Transition-based dependency parsing with stack long short-term memory. *arXiv preprint arXiv:1505.08075*, 2015.
13. ALBERTI, Chris; WEISS, David; PETROV, Slav. Improved transition-based parsing and tagging with neural networks. 2015.
14. MARCUS, Mitchell P.; MARCINKIEWICZ, Mary Ann; SANTORINI, Beatrice. Building a large annotated corpus of English: The Penn Treebank. *Computational linguistics*, 1993, 19.2: 313-330.
15. COLLINS, Michael. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In: *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*. Association for Computational Linguistics, 2002. p. 1-8.
16. TAYLOR, Ann; MARCUS, Mitchell; SANTORINI, Beatrice. The Penn treebank: an overview. In: *Treebanks*. Springer Netherlands, 2003. p. 5-22.
17. ZHOU, Hao, et al. A Neural Probabilistic Structured-Prediction Model for Transition-Based Dependency Parsing. In: *ACL (1)*. 2015. p. 1213-1222.
18. BÖHMOVÁ, Alena, et al. The Prague dependency treebank. In: *Treebanks*. Springer Netherlands, 2003. p. 103-127.
19. HOVY, Eduard, et al. OntoNotes: the 90% solution. In: *Proceedings of the human language technology conference of the NAACL, Companion Volume: Short Papers*. Association for Computational Linguistics, 2006. p. 57-60.
20. WEISCHEDEL, Ralph, et al. OntoNotes: A large training corpus for enhanced processing. *Handbook of Natural Language Processing and Machine Translation. Springer*, 2011.
21. JURAFSKY, Dan; MARTIN, James H. *Speech and language processing*. Pearson, 2014.
22. NIVRE, Joakim; MCDONALD, Ryan T. Integrating Graph-Based and Transition-Based Dependency Parsers. In: *ACL*. 2008. p. 950-958.
23. NIVRE, Joakim. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 2008, 34.4: 513-553.
24. SILVEIRA, Natalia, et al. A Gold Standard Dependency Corpus for English. In: *LREC*. 2014. p. 2897-2904.
25. ZHANG, Yue; CLARK, Stephen. A tale of two parsers: investigating and combining graph-based and transition-based dependency parsing using beam-search. In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2008. p. 562-571.
26. RUSSELL, Stuart; NORVIG, Peter; INTELLIGENCE, Artificial. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 1995, 25: 27.
27. HE, He; DAUMÉ III, Hal; EISNER, Jason. Dynamic Feature Selection for Dependency Parsing. In: *EMNLP*. 2013. p. 1455-1464.
28. GOLDBERG, Yoav; NIVRE, Joakim. Training deterministic parsers with non-deterministic oracles. *Transactions of the association for Computational Linguistics*, 2013, 1: 403-414.
29. CHOLLET, Fran\c{c}ois. Keras. In keras.io [interactive]. 2015. [Previewed 2017- 03-01]. Access through internet: https://keras.io/.
30. MIKOLOV, Tomas, et al. Distributed representations of words and phrases and their compositionality. In: *Advances in neural information processing systems*. 2013. p. 3111-3119.
31. REHUREK, Radim; SOJKA, Petr. Software framework for topic modelling with large corpora. In: *In Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. 2010.
32. RUBENSTEIN, Herbert; GOODENOUGH, John B. Contextual correlates of synonymy. *Communications of the ACM*, 1965, 8.10: 627-633.