NLP Assignment 4

Group 48 - Daimy van Loo (2852535), Kushnava Singha (2872230), Alexia Spinei (2692849)

May 26, 2025

1 Group Agreement Reflection

1.1 Decision Making

The decisions in our group are mostly made in the first meeting where we acknowledge the work, go through the work division, and set up expectations for this assignment. We note this down so everyone is on the same page at the start, and when things come up in between we discuss this either via Whatsapp or in another meeting. Honestly this was already very effective and efficient to us. For this assignment one change we should probably implement is to discuss a bit more during the course of the assignment, as the assignment is quite a bit bigger and more dependent on each other.

1.2 Group Work

From assignment A2, we learned that regularly updating each other on our progress and any issues we face helps us better manage our time before deadlines and allows us to offer more support if necessary. Thus, we think putting more attention toward this approach will help us be more efficient and effective, leading to a cohesive final report.

1.3 Timeline and Goal

Timeline: Part 2 & 3 - Daimy, aim to have this done by 14th may. Part 4 - Kush, aim to have this done by 16th may. Revise if this is not attainable and discuss with the group Part 5 - Alexia, aim to have a & b done by 20th, c by 25th (since it depends on part 4). Bonus - Daimy, aim to have this done by 23rd. This timeline shows our contributions to the assignment.

Our main learning goal for this assignment is understanding syntactic structure. We went into this assignment with little syntactic knowledge and did not know what to expect exactly. We can say that the assignment actually gave us more understanding on this topic, illustrating the straight-forward, but most importantly challenging parts of syntax in NLP. Therefore, we believe we have reached our learning goal, and hope to show this in our answers and implementations.

2 Machine Learning & Neural Networks

2.1 Stochastic Gradient Descent

Why is stochastic gradient descent useful for dependency parsing? Stochastic gradient descent has the same basic workings as gradient descent, with one important difference: it only uses one data point at a time to calculate the loss and necessary parameter updates. This results in much less required compute (higher efficiency), but a more fluctuating convergence, since parameter updates are only an approximation of true updates needed for the whole data. As there is often much data necessary for dependency parsing (millions/billions of tokens), it is essential that we can efficiently train a model on such data. Using stochastic gradient descent is therefore useful because of its efficiency.

2.2 Adam Optimizer

How does using m reduce variance of updates and why is this helpful to learning? By keeping a rolling average of the gradients, the next gradient is always highly dependent on the previous gradient. A β_1 of .9 would mean that 90% of the next gradient is based on the previous gradients, and only 10% is dependent on the actual gradient of the current loss. As the loss has a lot of variance in stochastic gradient descent, this method will significantly reduce this variance. This is helpful to learning because it helps the model keep updates in the right direction, which will lead to faster convergence.

Which parameters get larger updates by dividing by $\sqrt{v}$ and why is this helpful to learning? By dividing the update by $\sqrt{v}$, parameters with smaller previous gradients will get larger updates. This may be helpful to learning as these parameters have not received much attention by the model yet, since their gradients have been so small. By adaptively seeking out such parameters, a model may avoid being too focused on certain parameters, which have already received significant updates. This ensures all parameters get the opportunity to get updated, this may be very useful in avoiding local minima/optima.

2.3 Dropout

Why is dropout applied during training? As is mentioned, dropout is a regularization technique. Such techniques are used to control for overfitting of the model. Overfit means that the model starts to fit too much on training data, not only including the general signal, but also noise that is specific to this data. If this happens, the model will start to perform worse on validation and testing data, because the train-specific noise is not present there. Thus, if you want to control for overfitting, this needs to be done during training, which is why dropout is a regularization technique that should be applied during training.

Why should dropout not be applied during evaluation? The aim of evaluation is to assess the final performance of a model. As is mentioned on the paper (https://www.cs.toronto.edu/ hinton/absps/JMLRdropout.pdf), during testing you are basically aiming to average all predictions from the many 'thinned' networks that result from using dropout during training. However, as this number of networks exponentially increases with the number of nodes in the network, it is not feasible to do this. Thus, a way of approximating these averaged predictions is to use a final 'unthinned' network, where the weights of the nodes are multiplied by p_drop.

3 Neural Transition- Based Dependency Parsing

Stack	**Buffer**	**New Dependency**	**Transition**
[ROOT]	[I, attended, lectures, in, the, NLP, class]	–	Initial Configuration
[ROOT, I]	[attended, lectures, in, the, NLP, class]	–	SHIFT
[ROOT, I, attended]	[lectures, in, the, NLP, class]	–	SHIFT
[ROOT, attended]	[lectures, in, the, NLP, class]	attended $\rightarrow$ I	LEFT-ARC
[ROOT, attended, lectures]	[in, the, NLP, class]	–	SHIFT
[ROOT, attended, lectures, in]	[the, NLP, class]	–	SHIFT
[ROOT, attended, lectures, in, the]	[NLP, class]	–	SHIFT
[ROOT, attended, lectures, in, the, NLP]	[class]	–	SHIFT
[ROOT, attended, lectures, in, the, NLP, class]	[]	–	SHIFT
[ROOT, attended, lectures, in, the, class]	[]	class $\rightarrow$ NLP	LEFT-ARC
[ROOT, attended, lectures, in, class]	[]	class $\rightarrow$ the	LEFT-ARC
[ROOT, attended, lectures, in]	[]	in $\rightarrow$ class	RIGHT-ARC
[ROOT, attended, lectures]	[]	lectures $\rightarrow$ in	RIGHT-ARC
[ROOT, attended]	[]	attended $\rightarrow$ lectures	RIGHT-ARC
[ROOT]	[]	ROOT $\rightarrow$ attended	RIGHT-ARC

Table 1: Parsing steps for "I attended lectures in the NLP class."

A sentence containing n words will be parsed in how many steps (in terms of n)? Briefly explain in 1–2 sentences why. The algorithm will take $2n$ steps in order to parse through a sentence containing n words. This is because each word is first shifted from the Buffer to the Stack using the `SHIFT` transition which takes n steps and subsequently takes n steps to build dependencies using transitions (`LEFT-ARC` or `RIGHT-ARC`), leading to $2n$ steps in total. This is validated by Table 1 which takes 14 steps to parse a 7 word long sentence.

Inspect the data you will be using manually. What format is it in? How is it separated? Explain how the data is labeled already for your use. The data is in a CONLL-U standard format which annotates the data at a sentence level and at a token level. Each token is present in a separate line and contains 10 single tab separated columns. For example

```
1 Influential _ ADJ JJ _ 2 amod _ _
2 members _ NOUN NNS _ 10 nsubj _ _
```

Influential is the first token which is represented by 1 under the TokenID column, it is an adjective which is indicated by "ADJ" under the 4th column. The token influential also modifies the noun "members" (Token ID 2), which is represented by the dependency relation 'amod' under the 8th column. The POS tags and the dependency relation labelling in the data makes it optimal for parsing tasks.

Report the best UAS your model achieves on the dev set and the UAS it achieves on the test set in your write-up. Why is UAS a useful metric for evaluating dependency parsing? Cite specific examples where the UAS metric does and does not provide meaningful insight into the performance of your parser. The best Unlabeled Attachment Score that was achieved by our model on the dev set is 71.24 and on the test set is 89.13. The UAS score measures the percentage of words that are assigned the correct syntactic head, without taking the dependency label into consideration. Which helps us to assess how well the model captures sentence structures without considering errors in dependency labeling. From the previous example if a model correctly identifies that members is dependent on attended but mislabels the relation as "obj" instead of "nsubj", the UAS would still count it as correct.

```
17  new       _  ADJ  JJ  _  20  amod  _  _
20  agency    _  NOUN NN  _  22  nsubj _  _
```

In this example "new" is correctly attached to "agency" with the amod label indicating it as an adjectival modifier. However, if a parse mistakenly labels it as advmod, the attachment would still be considered correct under UAS even though the function role is mislabelled. In such cases relying solely on UAS may give an erroneous sense of parsing accuracy.

4 Error Analysis

a)

i.

Error type: Verb Phrase Attachment Error
Incorrect dependency: acquisition $\rightarrow$ citing
Correct dependency: blocked $\rightarrow$ citing
Explanations: "citing concerns" is a verb phrase that explains the action "blocked", and not the object "acquisition".

ii.

Error type: Modifier Attachment Error
Incorrect dependency: left $\rightarrow$ early
Correct dependency: afternoon $\rightarrow$ early
Explanations: In this case, "early" modifies "afternoon" because it tell us in which part of the afternoon the action of leaving occurred.

iii.

Error type: Prepositional Phrase Attachment Error
Incorrect dependency: declined $\rightarrow$ decision
Correct dependency: reasons $\rightarrow$ decision
Explanations: ”decision” should be attached to the head word ”reasons” because it is part of the phrase ”reasons for the government decision”, and it’s not directly related to the word ”declined”

iv.

Error type: Coordination Attachment Error
Incorrect dependency: affects $\rightarrow$ one
Correct dependency: plants $\rightarrow$ one
Explanations: ”One” is the second conjunct in a coordinated noun phrase and actually refers to another ”plant”.

b)

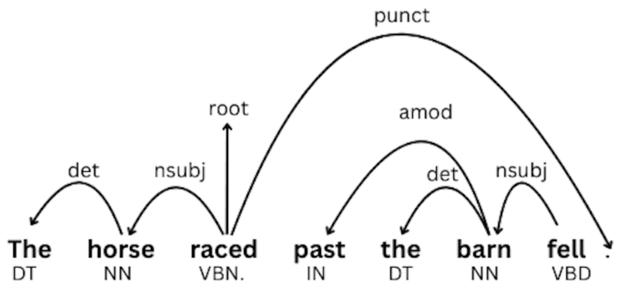

Figure 1: CoreNLP parse

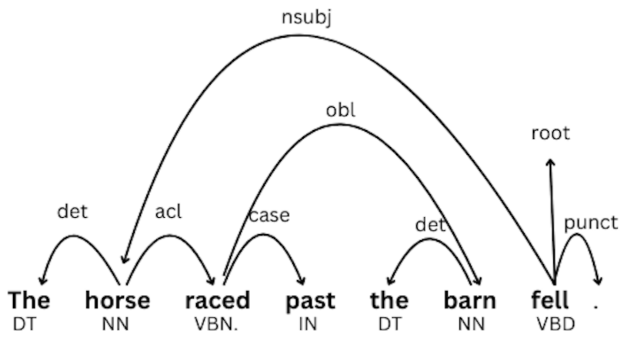

Figure 2: Correct parse

We can see the difference between the correct parse and the CoreNLP parse of the sentence “The horse raced past the barn fell.” in Figure 1 and Figure 2. In the correct interpretation, “raced past the barn” is a relative clause that modifies “horse”, and only “fell” is encoded as the main verb of the sentence. CoreNLP’s parser makes quick decisions and assumes that “raced” is the main verb and “horse” is its subject. This leads to more errors, such as labeling ’barn’ as the subject of ’fell’ and labeling ’past’ as an adjective modifier instead of a preposition. These errors come from CoreNLP’s lack of ability to review its decisions, making garden path sentences like this difficult to handle correctly.

In order to correctly parse the sentence “The horse raced past the barn fell,” we need to make a transition-based dependency deylay its early attachments and consider multiple possible interpretations as it processes the sentence. A typical parser assigns “raced” as the main verb and “horse” as its subject before it has seen the full sentence, which immediately leads to incorrect dependencies. To avoid this, the parser could use techniques like beam search, which keeps the top k most likely parse states at each step, allowing it to recover if an early decision turns out to be wrong. It could also be trained on examples of reduced relative clauses and use POS tags, such as in our case, recognizing 'raced' as a past participle (VBN) to support delayed attachment and accurate interpretation of the clause.

c)

Using POS tags as features in a parser can help by providing a small set of reusable labels that the parser can use to encode the most usual syntactic behavior of each word. This lets the parser use these labels to generalize patterns, rather than memorizing every possible word combination, making it better at handling unseen word sequences and ambiguous syntactic cases. Since the parser can make only three simple decisions based on the top words in the stack and buffer, POS tags can help provide a clearer signal about how those words are likely to attach, which leads to more accurate transition choices.

In our parser, we saw a clear example of why POS tags are helpful with "Take." In the dataset, "Take" is tagged correctly (VBP), meaning it's a finite verb (used in commands or present tense). However, the model incorrectly tagged it (VB), which usually marks the infinitive form (such as 'to take'). This small change leads the parser to behave very differently. When the tag is VBP, and the buffer contains a determiner (like "the"), the parser correctly recognizes a common structure such as "Take the medicine" and chooses SHIFT, allowing "the" and "medicine" to have their own dependency and be attached later as the object of the verb. But if the tag is VB, the parser doesn't recognize this pattern, and can incorrectly choose LEFT-ARC, making "the" the head of "Take." This breaks the tree and misrepresents the sentence structure. So even though the words are the same, the POS tag changes the parsing process, even though the sequence of words is exactly the same.

5 Bonus - Parsing Beyond English

For this assignment we chose the Dutch language. The Dutch language has some potential linguistic phenomena that may present challenges for a transition-based dependency parses, two of which will be explored in this section.

5.1 Flexible Word Order

5.1.1 The Phenomenon

A very common phenomenon in the dutch language is that of a switching order of certain words. For this assignment, we will focus on subject-verb-object (SVO) vs. subject-object-verb (SOV). In Dutch, SVO is the usual way of generating standard present or past tense sentences, such examples are shown in glosses (1) and (2).

(1)

Ik	*gooide*	*een*	*bal*
1SG	throw.PST	INDF	ball
I	threw	a	ball

'I threw a ball'

(2)

Hij	*neemt*	*een*	*slok*
3SG	take.PRS	INDF	sip
He	takes	a	sip

'He takes a sip'

However, when it comes to any compound tense (past perfect, present perfect, future perfect) in Dutch, the word order switches to SOV. See gloss (3) and (4) for two examples of this switched word order. Gloss (4) shows a special case of the present progressive tense, which indicates the action is taking place now. In Enlish, this is often denoted by the suffix -ing to a verb accompanied by the verb 'be' (is raining, are biking, am running). In Dutch, the words 'aan het' are put before the verb to indicate that something is currently happening.

(3)

Ik	*heb*	*een*	*bal*	*gegooid*
1SG	AUX.PRS	INDF	ball	throw.PTCP
I	have	a	ball	thrown

'I have thrown a ball'

(4)

Hij	*is*	*een*	*slok*	*aan*	*het*	*nemen*
3SG	AUX.PRS	INDF	sip	PROG	DEF	take.INF
He	is	a	sip	on	the	taking

'He is taking a sip'

The phenomenon also occurs for other cases in Dutch, e.g. with subordinate clauses as shown in gloss (5). However, for the rest of this section, we will simply focus on two past tense examples.

(5)

Omdat	*ik*	*een*	*bal*	*gooide*
because	1SG	INDF	ball	throw.PST
Because	I	a	ball	threw

'Because I threw a ball'

5.1.2 Comparing Dependency Trees

Figure 3 and 4 show the gold Universal Dependency trees I drew for the sentences "Ik gooide een bal" (SVO) and "Ik heb een bal gegooid" (SOV).

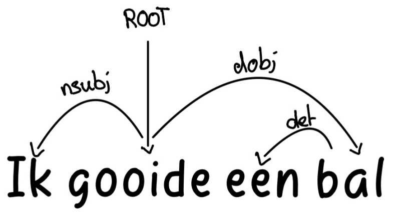

Figure 3: The gold tree for the sentence "I threw a ball" in Dutch.

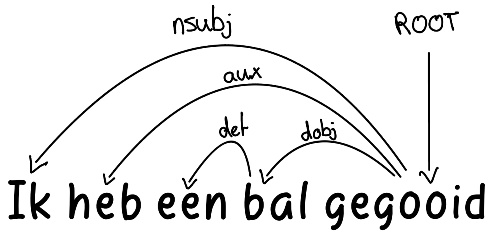

Figure 4: The gold tree for the sentence "I have thrown a ball" in Dutch

We chose to compare the gold dependency tree to the existing dependency parser in SpaCy, pre-trained for the Dutch language (`nl_core_news_md`), implemented in python. This dependency parser uses an arc-eager transition-based parsing approach (https://spacy.io/api/dependencyparser). Interestingly, the dependency trees result in exactly the same trees as the gold trees above. To avoid redundancy, the dependency parser trees are therefore not separately created again.

5.1.3 Why does it challenge a transition-based parser?

The fact that the transition-based parser does not make any mistakes here, is interesting because "ik heb een bal" is also a valid sentence which literally translates to "I have a ball". A very simple transition-based parser could therefore create a subject and object like this: ik ← heb, heb → bal. This type of sentence is much more similar to the simple past tense sentence we show in figure 1. However, as we dive deeper into the working of a neural transition-based dependency parser, it makes sense that the parser does not make these mistakes. First, a neural transition-based parser is able to use context in order to learn when a transition (e.g. left-ARC/right-ARC) is appropriate. Thus, the parser does not have to assign a dependency to two words immediately if it seems fitting. Second, this parser is trained specifically on a huge base of dutch written texts. Since the SOV word-order is very common in the Dutch language for certain tenses as mentioned before, this parser has seen enough of these sentences in it's training data to be able to accurately learn the patterns. Combining these two points, we can conclude that the parser has likely learn to prefer the SHIFT transition over a left- or right-ARC transition when encountering possible Dutch auxillary verbs like 'heb', 'ben', and 'kan'.

5.1.4 Implications of Such Challenges

As we have seen in this section, the differing word order between SVO and SOV in Dutch actually does not seem that big of a challenge. We have seen that a simple neural transition-based parser can accurately parse such sentences. We think there is one very important factor here, that is essential for this performance; there is enough data from the Dutch language. Although the Netherlands is a small country with not that many residents (compared to big countries like the USA or China), it is still quite influential in other aspects. The Dutch language therefore has an abundance of textual data, in research as well as other domains like news, law, and blog-posts. This is essential if you want any NLP system to be able to work accurately for a given language. Therefore, we do not think this particular phenomenon would cause serious challenges in most NLP tasks.

5.2 Ellipsis

5.2.1 The Phenomenon

An ellipsis is a common phenomenon, not just in the Dutch language. A linguistic ellipsis occurs when part of a sentence is omitted because it can be inferred from context (other previous or next words) what is meant. Gloss (6) shows a clear example of a simple common ellipsis, while gloss (7) shows the way it should be (and often implicitly is) interpreted by humans.

(6)

Ik	*gooi*	*een*	*bal*	*en*	*hij*	*een*	*boek*
1SG	throw.PRS	INDF	ball	and	3SG.M	INDF	book
I	throw	a	ball	and	he	a	book

‘I throw a ball and he a book.’

(7)

Ik	*gooi*	*een*	*bal*	*en*	*hij*	*gooit*	*een*	*boek*
1SG	throw.PRS	INDF	ball	and	3SG.M	throw.PRS	INDF	book
I	throw	a	ball	and	he	throws	a	book

‘I throw a ball and he throws a book.’

5.2.2 Comparing Dependency Trees

An ellipsis basically results in a somewhat incomplete sentence. In the given example, the crucial verb ’throws’ is omitted, which would receive an subject and object otherwise. After speaking to Dr. Donatelli in class, she advised me it is often easiest to just pretend the word is there when drawing dependency trees. We will apply this approach, going more in depth as to why this approach is reasonable here when we compare the gold tree to the one from an existing dependency parses. Figure 5 shows the gold Unisversal Dependency tree the way humans would interpret an elliptic sentence (implicitly introducing the omitted sentence). Figure 6 shows the dependency created by the SpaCy dependency parser. Discrepancies in the SpaCy tree are shown in red.

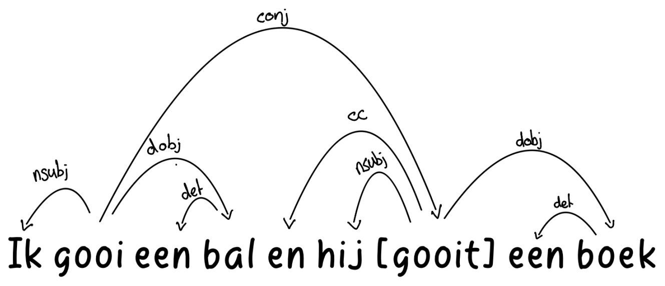

Figure 5: The gold tree for the sentence ”I throw a ball and he [throws] a book” in Dutch.

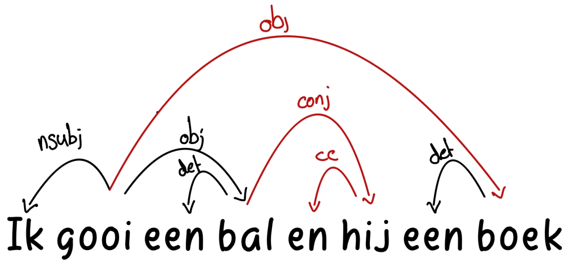

Figure 6: The dependency parser tree for the sentence ”I throw a ball and he a book” in Dutch.

As expected, there are clear discrepancies between these two trees. Is the SpaCy dependency parser necessarily wrong? We do not think so. It is simply doing the best with what is given, and the available transitions it can make while parsing. The discrepancies show a very coherent pattern with the gold tree: there is some sort of conjunction, and it is related to the verb we have already seen. The dependency parser simply does not have the tools to process this sentence in the way a human can (i.e. implicitly adding the omitted word). As the most important word from the second part of the sentence (”throws”) is not present for the parser, it cannot create any dependencies on it.

5.2.3 Why does it challenge a transition-based parser?

As mentioned in the section above, a dependency parser simply does not have all the tools to parse an elliptic sentence the way humans do. Even if it is able to recognize the correct dependencies (like it did for our example), it does not have all the words to correctly connect these dependencies. For example, the SpaCy parser is limited to the shift operation where it can move one word from the buffer to the stack. The only words that will be added to the buffer are those that are initially in the sentence. In order for the parser to be able to move toward a parse that looks more like the gold tree, a 'word addition' operation would have to be added, where the parser can also add words to the buffer. In addition, the SpaCy parser is limited to arc-eager transitions (e.g. ARC-left and ARC-right). Special to arc-eager transitions is that a dependency can be added to any two words that are 'visible', they do not have to be next to each other on the stack. However, as we said before that the parser can only ever work with words that are explicitly in the sentence, it naturally follows that there can only ever be dependencies between these words. Thus, a transition-based parser may add dependencies between words that you would normally not expect (e.g. bal $\rightarrow$ hij), as it may recognize a certain dependency but it lacks the proper word to connect it.

5.2.4 Implications of Such Challenges

The above section seems to indicate that the ellipsis phenomenon may cause issues in NLP tasks. However, the fact that the model was able to correctly extract the right dependencies, but it was just not able to attach the to the words we would do, may indicate otherwise. We would argue that the challenges related to this phenomenon would actually not cause significant challenges in most NLP tasks for two reasons. First, as we already explained for the previous phenomenon, the Dutch language is quite extensively documented and readily available online, ensuring any NLP systems would likely have enough data to accurately learn these patterns as well. Second, the fact that the ellipsis is also a common phenomenon in English, and even occurs in similar patterns, further ensures that NLP systems have ample data to learn to correctly handle this phenomenon. Since, as we know, research around NLP using English data is plentiful. Thus, we argue that again, the challenges this phenomenon poses to this dependency parser would not cause serious challenges in most NLP tasks.