Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Master Thesis in Computer Engineering

Dynamic Oracle for the Arc-Standard Parser: from Approximation to Reinforcement Learning

Master Candidate

Pietro Picardi

Student ID 2052415

Supervisor

Prof. Giorgio Satta

University of Padova

Co-supervisor

Prof. Gian Antonio Susto

University of Padova

Co-supervisor

Niccolò Turcato, dott.

University of Padova

Academic Year
2023/2024

To my family
and friends

Abstract

Dependency parsing is one of the foundational tasks in Natural Language Processing and forms a backbone for syntactic analysis and language understanding. The Arc-Standard transition based parser has largely been adopted to solve this task, due to its simplicity and effectiveness, but the computational demands that are attached with its dynamic oracle present very crucial challenges when dealing with non-projective data.

This thesis deals with the computational inefficiencies of the dynamic oracle for the Arc-Standard parser. Firstly, an overview is given of the main approaches to this task, then a more in depth analysis of transition-based parsers and their oracles follows, in particular detailing the motivations and strengths associated with the dynamic oracle. Moving on, our research follows the academic literature reporting the arc decomposition property and its key role in deriving exact linear dynamic oracles: after explaining why the Arc-Standard parser doesn't present this theoretical feature, an exact, but computationally expensive, dynamic oracle is implemented. The thesis continues with the implementation of a linear approximate dynamic oracle, trying to cut down computational complexity while preserving parsing accuracy.

Guided by these findings, this dissertation considers using the reinforcement learning approach in reformulating the dynamic oracle. Applying machine learning techniques to adapt parsing decisions dynamically, given a learned model, can increase both the efficiency and the accuracy of this process. In this direction, our contribution is a first attempt on reducing computational demands of the exact dynamic oracle to achieve performance comparable to those original formulations.

The thesis presents a blend of theoretical discussions, empirical evaluations, and practical implementations for examining how well provided dynamic oracle optimizations work. Experimental results illuminate trade-offs of computational efficiency against parsing accuracy through the stages of our enhancement process. In addition, this research offers insights into the field of dependency parsing in that it suggests a reinforcement learning framework tailored specifically to address the challenges presented by the dynamic oracle of the Arc-Standard parser.

Sommario

Sommario

L'analisi delle dipendenze è uno dei compiti fondamentali nel Natural Language Processing e costituisce un pilastro per l'analisi sintattica e la comprensione del linguaggio. Il parser basato sulle transizioni Arc-Standard è stato ampiamente adottato per via della sua semplicità ed efficacia, ma le richieste computazionali che sono associate al suo oracolo dinamico presentano sfide molto importanti quando si ha a che fare con dati non proiettivi.

Questa tesi riguarda le inefficienze computazionali dell'oracolo dinamico per il parser Arc-Standard. In primo luogo viene fornita una panoramica dei principali approcci a questo compito, segue un'analisi più approfondita dei parser basati su transizioni e dei loro oracoli, in particolare dettagliando le motivazioni e i punti di forza associati all'oracolo dinamico. Proseguendo, la nostra ricerca segue la letteratura accademica che riporta la proprietà di decomposizione dell'arco e il suo ruolo chiave nel derivare oracoli dinamici lineari esatti: dopo aver spiegato perché il parser Arc-Standard non presenta questa caratteristica teorica, viene implementato un oracolo dinamico esatto, ma computazionalmente costoso. La tesi continua con l'implementazione di un oracolo dinamico lineare approssimato, cercando di ridurre la complessità computazionale pur preservando l'accuratezza del parsing.

A questo punto, la dissertazione considera l'utilizzo dell'approccio di apprendimento per rinforzo nella riformulazione dell'oracolo dinamico. Applicare tecniche di machine learning per adattare dinamicamente le decisioni di parsing, dato un modello appreso, può aumentare sia l'efficienza che l'accuratezza di questo processo. In questa direzione, il nostro contributo è un primo tentativo di ridurre le richieste computazionali dell'oracolo dinamico esatto per ottenere prestazioni paragonabili a quelle delle formulazioni originali.

La tesi presenta una miscela di discussioni teoriche, valutazioni empiriche e implementazioni pratiche per esaminare quanto bene funzionino le ottimizzazioni dell'oracolo dinamico fornite. I risultati sperimentali mostrano i compromessi tra efficienza computazionale e accuratezza del parsing attraverso le fasi del nostro processo di miglioramento. Inoltre, questa ricerca offre intuizioni nel campo dell'analisi delle dipendenze in quanto suggerisce un framework di apprendimento per rinforzo su misura per affrontare le sfide presentate dall'oracolo dinamico del parser Arc-Standard.

Contents

List of Figures

List of Tables

List of Algorithms

List of Code Snippets

List of Acronyms

BiLSTM Bidirectional LSTM

DP Dynamic Programming

DDQN Dueling Deep Q-Network

DQN Deep Q-Network

DRL Deep Reinforcement Learning

GPI Generalized Policy Iteration

MDP Markov Decision Process

MC Monte Carlo

MLP Multi-Layer Perceptron

NLP Natural Language Processing

PPRB Proportional Prioritized Replay Buffer

RL Reinforcement Learning

TD Temporal Difference

UAS Unlabeled Attachment Score

UD Universal Dependencies

1 Introduction

This section presents an introduction to Natural Language Processing (NLP), a field with a focus on areas of intersection with computer science, linguistics, and artificial intelligence. The following content is based on *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition* by Jurafsky et al. (2009) [7]. Specifically, dependency parsing is presented as one of the first task developed in NLP. From being a syntactic analysis tool to a possible enhancing component in advanced applications like machine translation and sentiment analysis, dependency parsing is a foundational task of NLP. The chapter delves into the technicalities of dependency parsing, discussing its representation as a directed graph, the importance of grammatical relations for forming dependency structures, and the idea of projectivity of a sentence. These discussions underscore the role of dependency parsing in the larger context of modern NLP, underpinning its practical applications and theoretical implications in language understanding and processing.

1.1 Natural Language Processing

NLP is located at the intersection of computer science, linguistics, and artificial intelligence, dealing with the bridging of human language and machine understanding. The evolution of NLP can be traced from the mid-20th century, when researchers began to explore the computational aspects of language, to our days: over the years, NLP has increased its relevance in our modern world where large sets of information are encoded in the written text. The explosion in digital content, social media, and online communication has caused written language to be one of the richest sources of data. The emergence of Language Models (LMs) and, particularly, the more popular development of Large Language Models (LLMs) have also dramatically changed the sphere of NLP. Large Language Models (LLMs), of which OpenAI's GPT-3 is one, have demonstrated unprecedented abilities in both understanding and producing human-like text.

Historically, one of the first tasks in the NLP domain is dependency parsing. This task was originally conceived to represent the syntactic relationships between the words in the sentence; however dependency parsing underwent renewed importance in modern applications. Unlike its traditional role of syntactic analysis, modern dependency parsing is instrumental in various cutting-edge NLP tasks, such as machine translation, sentiment analysis, and question answering. The ability of dependency parsing to capture the hierarchical and grammatical structures of language has proven invaluable in enhancing the performance of these tasks. This paradigm shift in the way dependency parsing is applied underlines its adaptability and continued relevance in a dynamic landscape such as the one of natural language understanding.

1.2 Dependency Parsing

In NLP, the process of dependency parsing refers to the identification of syntactic structures in a sentence and has historically played an intermediate role for more in-depth language processing. Modern NLP does not explicitly employ structures from the parse into neural language models, but there are new methods for using them.

A summary of the major functions of the linguistic structure in modern NLP is presented. Firstly, it acts as a tool for the interpretation of neural networks: understanding that specific layers or neurons may be computing syntactic structures helps unveiling the "black box" nature of language models. Secondly, the structure of language does provide a toolkit for social scientific studies of text in a practical sense. It aids in the measurement of attitudes by distinguishing relationships between words, such as adjectives modifying nouns or implied metaphors in use.

Moreover, detailed semantic structures have practical applications, such as in legal contracts, where identifying specific clauses with particular meanings is important. Word sense labels are emphasized due to the fact that they serve the purpose of safeguarding corpus studies from measuring the wrong word sense. Finally, linguistic structure matters in answering larger questions about the nature of language, specifically those dealing with its changes over time or across individuals and therefore require the ability to parse entire documents from different time periods.

The dependency formalism describes the syntactic structure of a sentence in terms of directed binary grammatical relations that hold among words. The relations are represented above the sentence, in the form of labelled arcs from heads to dependents, resulting in a typed dependency structure with labels drawn from a fixed inventory of grammatical relations (unlabelled depedency trees are also used). Explicitly, the ROOT node marks the root of the tree and signals the head of the whole structure. This policy of direct encoding of key information in head-dependent relationships simplifies complex phrase-structure parses.

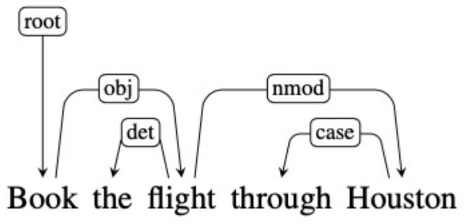


Figure 1.1: Dependency parsing tree

These dependency structures are based on the traditional conception of grammatical relations in language. Binary relations consist of a head (central organizing word) and a dependent (modifier): these relations present the feature of making the association between heads and their immediate dependents obvious, providing important syntactic information. In a dependency grammar, one specifies these head-dependent pairs, but one also classifies the grammatical relations or functions which the dependent serves in relation to its head. This classification includes well-known concepts like subject, direct object, and indirect object: while these notions in English often correlate with sentence position and constituent type, they are not strictly determined by them. In languages where word order is considerably less flexible, phrase-based constituent syntax gives little explicit guidance to the understanding of the syntactic structure, therefore the information encoded in grammatical relations becomes crucial for understanding syntactic structure.

1.2.1 Dependency Formalisms

A dependency structure can be defined as a directed graph G = (V, A), where V is a set of vertices and A is a set of ordered pairs of vertices, referred to as arcs. In the context of linguistic dependency parsing, V typically corresponds to the words in a sentence, but it can also include elements like punctuation or morphological components in languages with complex morphology.

Various grammatical theories or formalisms might impose additional conditions on these dependency structures. Connectivity, existence of a ROOT node, and acyclicity (or planarity) are typical restrictions. In this context, the most important computational constraint is that of rooted trees, which is going to play a crucial role in parsing methods. A dependency tree has to satisfy the following conditions:

1. One unique designated ROOT node with no incoming arcs exists.
2. Except for the ROOT node, each vertex has exactly one incoming arc.
3. A unique path from the ROOT node to each vertex in V exists.

Such constraints ensure that each word in the sentence has a single head, and a definite relationship that specifies who is the head of whom. They also make sure that the dependency structure is connected and that there is a unique ROOT node from which a directed path can be followed to reach every word in the sentence. This greatly simplifies both the analysis of and computation on the linguistic dependencies present in the natural language sentences.

1.2.2 Projectivity

The concept of projectivity introduces an additional constraint based on the word order in the input. An arc from a head to a dependent is called projective if there is a path from the head to every word lying between the head and the dependent in the sentence. Consequently, a dependency tree is deemed projective if all its constituent arcs are projective.

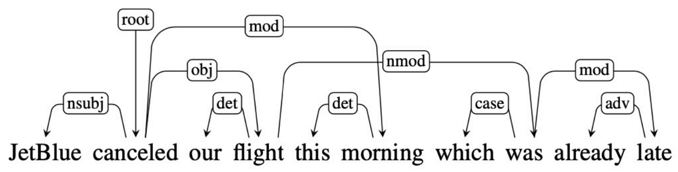

root
mod
obj
nmod
mod
nsubj
det
det
case
adv
JetBlue canceled our flight this morning which was already late

Figure 1.2: Non-projective sentence

Consider Figure 1.2: the arc from ′flight′ to its modifier ′was′ is non-projective, since there is no path from ′flight′ to the words ′this′ and ′morning′ which lie in between. Projectivity, or its absence (non-projectivity), can be seen from the tree diagrams: a dependency tree is projective if it can be drawn without any crossing edges. For instance, in order to link ′flight′ to its dependent ′was′, one would have to cross the arc that links ′morning′ to its head.

This concern with projectivity derives motivation from two related issues. First, widely used English dependency treebanks are often automatically derived from phrase-structure treebanks using head-finding rules that a priori result in projective trees; this means that if a non-projective example is met, a problem arises. Moreover, some parsing algorithms, especially those of the transition-based flavor, are restricted to producing only projective trees; thus, sentences with non-projective structures may suffer from parsing errors. This may be a reason for choosing graph-based parsing as a more flexible but also more computational expensive parsing strategy.

2 Background

This chapter analyzes the different techniques of dependency parsing that can be used within NLP. It starts with analyzing transition-based parsing, where the parsers build dependency trees using a stack and a buffer and a set of transitions, facing problems such as ambiguity in language and limitations of the oracle. Then we elaborate on graph-based parsing, where a score function is optimized to select the best dependency tree resulting in more accuracy in handling long sentences and non-projective structures, but with increased computational complexity. Lastly, a discussion on sequence-based parsing, which predicts grammatical relationships in a linear sequence using machine learning. This approach, efficient in handling input, is potentially not as effective in capturing long-range dependencies as the more holistic graph-based parsing. Each method presents unique advantages and drawbacks, underscoring the diverse strategies employed in parsing and in understanding natural language.

2.1 Transition Based Dependency Parsing

The transition-based parsing approach has a stack σ, a buffer of tokens β, and an oracle that dictates the next transitions that must be made to construct a dependency parse. The parser reads the sentence from left to right, moving words from the buffer to the stack, while the oracle decides transitions depending on the current configuration $c = (\sigma, \beta, A)$.

Two kinds of transition operator formalize the intuitive actions:

- **REDUCTION:** assert a head-dependent relation, removing the dependent word from the stack.
- **SHIFT:** remove a word from the input buffer, push it into the stack.

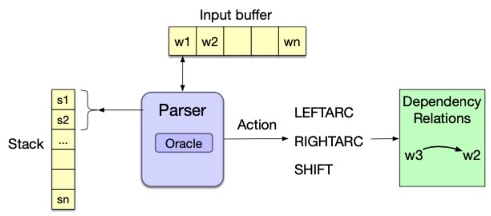


Figure 2.1: Transition-based parsing architecture

In the example below, these operators adhere to the Arc-Standard approach, asserting relations only between elements at the top of the stack: there are alternative transition systems that show different parsing behavior, but the Arc-Standard is pretty effective and easy to implement. The parser iterates over transitions while the buffer is not empty and the stack has only the ROOT node.

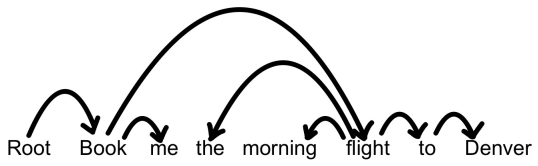


Figure 2.2: Gold parsing tree

Step	Stack	Buffer	Action	Relation Added
0	[ROOT]	[book, me, the, morning, flight, to, Denver]	SHIFT	
1	[ROOT, book]	[me, the, morning, flight, to, Denver]	SHIFT	
2	[ROOT, book, me]	[the, morning, flight, to, Denver]	RIGHTARC	(book→me)
3	[ROOT, book]	[the, morning, flight, to, Denver]	SHIFT	
4	[ROOT, book, the]	[morning, flight, to, Denver]	SHIFT	
5	[ROOT, book, the, morning]	[flight, to, Denver]	SHIFT	
6	[ROOT, book, the, morning, flight]	[to, Denver]	LEFTARC	(morning←flight)
7	[ROOT, book, the, flight]	[to, Denver]	LEFTARC	(the←flight)
8	[ROOT, book, flight]	[to, Denver]	SHIFT	
9	[ROOT, book, flight, to]	[Denver]	SHIFT	
10	[ROOT, book, flight, to, Denver]	[]	RIGHTARC	(to→Denver)
11	[ROOT, book, flight, to]	[]	RIGHTARC	(to←flight)
12	[ROOT, book, flight]	[]	RIGHTARC	(book→flight)
13	[ROOT, book]	[]	RIGHTARC	(ROOT→book)
14	[ROOT]	[]	Done	

Table 2.1: Parsing example

Consider the example in Tables 2.1 and 2.2: when examining transition sequences in dependency parsing, it's crucial to note that the given sequence is not always the sole path leading to a reasonable parse. Ambiguity in language may result in multiple transition sequences yielding the same parse. Different paths can lead to equally valid parses.

Step	Stack	Buffer	Action	Relation Added
0	[ROOT]	[book, me, the, morning, flight, to, Denver]	SHIFT	
1	[ROOT, book]	[me, the, morning, flight, to, Denver]	SHIFT	
2	[ROOT, book, me]	[the, morning, flight, to, Denver]	RIGHTARC	(book→me)
3	[ROOT, book]	[the, morning, flight, to, Denver]	SHIFT	
4	[ROOT, book, the]	[morning, flight, to, Denver]	SHIFT	
5	[ROOT, book, the, morning]	[flight, to, Denver]	SHIFT	
6	[ROOT, book, the, morning, flight]	[to, Denver]	SHIFT	
7	[ROOT, book, the, morning, flight, to]	[Denver]	SHIFT	
8	[ROOT, book, the, morning, flight, to, Denver]	[]	RIGHTARC	(to→Denver)
9	[ROOT, book, the, morning, flight, to]	[]	RIGHTARC	(to←flight)
10	[ROOT, book, the, morning, flight]	[]	LEFTARC	(morning←flight)
11	[ROOT, book, the, flight]	[]	LEFTARC	(the←flight)
12	[ROOT, book, flight]	[]	RIGHTARC	(book→flight)
13	[ROOT, book]	[]	RIGHTARC	(ROOT→book)
14	[ROOT]	[]	Done	

Table 2.2: Parsing example with alternative path

Moreover, in practice, there is no guarantee that the oracle will return the correct transition operator. Erroneous decisions of the oracle can happen in real applications. The greedy nature of transition-based methods means that incorrect choices at any step can lead to inaccurate parses, and the parser lacks the ability to backtrack and explore alternative choices. This greediness implies that it might not explore the entire search space, potentially missing alternative valid parses.

Furthermore, the examples are shown without labels on the dependency relations. To produce labeled trees, the LEFTARC and RIGHTARC operators can be parameterized with dependency labels (e.g., LEFTARC(NSUBJ) or RIGHTARC(OBJ)). This expands the set of transition operators, complicating the oracle's task as it now needs to choose from a larger set of operators.

Therefore, the flexibility and simplicity of transition-based parsing come with challenges related to ambiguity, the accuracy of the oracle, and the exploration of the search space. Improvement of robustness and generalization ability, including better capacity to handle ambiguity, precision of the oracle, and efficiency in covering the search space, are all required aspects for improvement of transition-based parsing in real-world applications.

2.2 Graph Based Dependency Parsing

Graph-based methods form an important family of algorithms for dependency parsing, which model the parsing task as a problem of maximizing some predefined score function so as to search for the dependency tree with the highest score (computed either through manually designed features or extracted via neural network). The various parameters are then adjusted by the system according to annotated data in the training process and use linguistic characteristics and relations to judge the proposed dependency trees. While transition-based parsing constructs parses step-by-step, graph-based methods traverse the entire possible dependency tree space to discover with an optimum score.

Graph-based parsers considerably differ from transition-based approaches. First, they exhibit superior accuracy, particularly advantageous for longer sentences, while transition-based parsers perform poorly whenever heads and dependents are significantly far from each other. Second, the graph-based parsers excel in handling non-projective structures, which is the desirable flexibility not exhibited within the transition-based approaches. This becomes very useful in languages with freer word orders, where non-projective structures are common.

Moreover, the decision-making process in graph-based methods is global rather than relying on local, greedy decisions. This comprehensive evaluation of entire trees contributes to improved accuracy in capturing syntactic structures. The search for the optimal dependency parse is framed as finding the maximum spanning tree in the space of possible trees, determined by the highest overall score.

However, the benefits of graph-based parsing are accompanied by a number of disadvantages as well. The enormous computational burden it places on searching the space of possible trees can be expensive, particularly in locating the maximum spanning tree. The scoring schemes also pose strenuous problems, demanding for meticulous capturing of syntactic relationships.

In conclusion, graph-based dependency parsing offers advantages in accuracy, non-projective structure handling, and global decision-making. However, these benefits come with challenges related to computational complexity and scoring. The choice between graph-based and transition-based methods depends on language characteristics and parsing task requirements.

2.3 Sequence Based Dependency Parsing

In sequence-based dependency parsing, the syntactic structure of a sentence is analyzed by predicting the grammatical relationships between words based on the sequential order of the input. Unlike transition-based parsing, in which there is a sequence of state transitions developing the dependency tree part by part, and graph-based parsing, which treats parsing as a global optimization task over the entire dependency graph, the sequence-based approach cares about capturing dependencies linearly. This method exploits machine learning or deep learning techniques with features like word embeddings and part-of-speech tags aiding the model in decision-making. The sequential approach here allows for efficient handling of the input sentences at hand. However, it may face challenges in capturing long-range dependencies and global interactions present in the sentence structure, a strength of graph-based methods. Transition-based parsing, on the other hand, strikes a balance by combining local transitions with a dynamic global view during parsing, making it adept at capturing both local and medium-range dependencies.

3 Parsers and Oracles

This chapter delves into transition-based parsing in NLP, examining different types of parsers and oracles. It introduces three parsers, the Arc-Standard, the Arc-Eager, and the Arc-Hybrid, each of them with their different sets of rules for building dependency trees. The chapter also describes the three main variants of oracles for transition-based systems: static, nondeterministic, and dynamic. Static oracles are rigid, with a fixed transition path and the lack of adaptability, which could be offered by the non-deterministic oracles, allowing many valid transitions and hence ambiguity. Finally, the dynamic oracles are able to adapt to the state of the parser and to handle complex syntactic structures effectively. The challenges of greedy parsers are emphasized, especially in error propagation, and the advantage of dynamic oracles in training for improved handling of linguistic phenomena is outlined.

3.1 The components of transition-based parsing

Transition-based parsing provides a framework that is very conceptual in nature, where an abstract machine is targeted towards the processing of sentences so that the corresponding parse trees can be build. It features a well-defined transition system (parser) that is made up of a set of configurations and a corresponding set of transitions. These are transitions that work on the configurations to shape the process of parsing. Applied to a sentence, the system starts from an initial configuration and then iteratively applies transitions. After a finite number of transitions, the system converges to a terminal configuration, from which a parse tree is derived. In this framework, the greedy parsing approach uses a classifier, trained through imitation learning, to decide at each configuration the most appropriate transition according to features extracted directly from the configuration itself and from an expert (oracle) to imitate. The efficacy of the transition-based parsing framework is contingent on the interplay between the chosen transition system, the definition of configurations and the specific set of transitions available. This parsing paradigm exemplifies a dynamic and iterative process that efficiently captures the syntactic structures inherent in natural language sentences.

3.2 Parser

In the domain of NLP, parsers help to unwrap these intricate grammatical sentence structures. Such computational entities are commissioned with the challenge of performing a syntactic analysis, helpful in the extraction of meaningful word-to-word relationships. The parsing process involves a delicate balance in guiding the parser through a series of actions that will result in properly building an accurate tree of dependencies. This introduction sets the stage for the exploration of three parsing models: Arc-Standard, Arc-Eager, and Arc-Hybrid. In the next few paragraphs, we shall look into these parsers in finer details, illustrating the specifics that keep them distinguished in the parsing approach. Every parsing example will refer to the gold dependency tree represented in Figure 3.1.

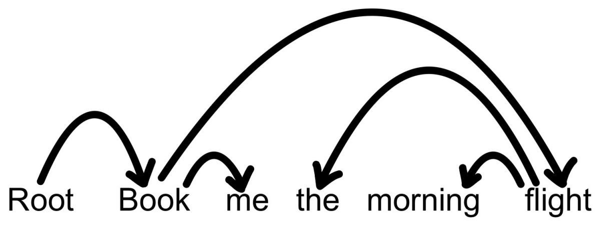


Figure 3.1: Dependency parsing tree

3.2.1 Arc-Standard

The Arc-Standard parser operates on a straightforward set of rules in dependency parsing, featuring three fundamental transition actions: SHIFT, LEFTARC, and RIGHTARC. The process begins with an initial configuration, where all words are in the buffer, the stack contains the ROOT, and there are no arcs.

- **LEFTARC:** Creates a dependency arc between the top stack word and the second-to-top stack word, and removes the second-to-top element from the stack.
$$\text{LEFT}_{\text{lb}}[(\sigma|s_1|s_0, \beta, A)] = (\sigma|s_0, \beta, A \cup \{(s_0, \text{lb}, s_1)\})$$
- **RIGHTARC:** Establishes a dependency arc between the second-to-top stack word and the top stack word, and removes the top element from the stack.
$$\text{RIGHT}_{\text{lb}}[(\sigma|s_1|s_0, \beta, A)] = (\sigma|s_1, \beta, A \cup \{(s_1, \text{lb}, s_0)\})$$
- **SHIFT:** Moves a word from the buffer to the top of the stack.
$$\text{SHIFT}[(\sigma, b|\beta, A)] = (\sigma|b, \beta, A)$$

This iterative process continues until a single word remains in the stack, and the buffer is empty, resulting in a valid dependency tree.

The Arc-Standard system imposes specific preconditions on the legality of transition actions. In this system, the LEFTARC transition is deemed legal only when the top element (s_1) on the stack is not the ROOT. Additionally, both LEFTARC and RIGHTARC transitions are considered legal only when the stack contains at least two elements. This set of conditions ensures a controlled and systematic construction of parse trees.

Step	Stack	Buffer	Action	Relation Added
0	[ROOT]	[book, me, the, morning, flight]	SHIFT	
1	[ROOT, book]	[me, the, morning, flight]	SHIFT	
2	[ROOT, book, me]	[the, morning, flight]	RIGHTARC	(book→me)
3	[ROOT, book]	[the, morning, flight]	SHIFT	
4	[ROOT, book, the]	[morning, flight]	SHIFT	
5	[ROOT, book, the, morning]	[flight]	SHIFT	
6	[ROOT, book, the, morning, flight]	[]	LEFTARC	(morning←flight)
7	[ROOT, book, the, flight]	[]	LEFTARC	(the←flight)
8	[ROOT, book, flight]	[]	RIGHTARC	(book→flight)
9	[ROOT, book]	[]	RIGHTARC	(ROOT→book)
10	[ROOT]	[]	Done	

Table 3.1: Arc-Standard parsing example

The Arc-Standard system follows a bottom-up tree-building approach: each word is allowed to accumulate all its dependents before it gets attached as modifier to its head. This bottom-up strategy indeed guarantees an organized and coherent representation of the syntactic structure. Importantly, the system does not enforce any ordering constraint on left and right dependents. The flexibility accounts for the different linguistic structures and ensures adaptability to the various ways in which dependencies may be expressed in a natural language sentences. The Arc-Standard system's adherence to these conditions and its bottom-up construction method contribute to its effectiveness in capturing the nuanced syntactic relationships within sentences.

However, the Arc-Standard parser has its limitations as well. It generalizes worse longer-range dependency relations than local ones and might face difficulty with some linguistic phenomena. The simplicity of this model, while an asset for efficiency, can be a limitation in capturing complex syntactic structures accurately.

On the other side, the Arc-Standard parser shows desirable qualities of computational efficiency and implementation simplicity. It is more suited for applications assigning priority in speed and ease of implementation, hence it is the preferred choice in practical application for several NLP tasks. While it may not excel in every linguistic nuance, its balance between accuracy and computational efficiency makes it a valuable tool in the arsenal of dependency parsers.

3.2.2 Arc-Eager

The Arc-Eager parser is another approach to dependency parsing that employs a set of rules to construct a syntactic tree. The initial configuration consists of an empty stack, a buffer containing all of the words with the ROOT as last element and no arcs.

The rules of the Arc-Eager parser are as follows:

- **LEFTARC:** Assert a head-dependent relation between the word at the front of the input buffer and the word at the top of the stack; pop the stack.

$$\text{LEFT}_{\text{lb}}[(\sigma|s, b|\beta, A)] = (\sigma, b|\beta, A \cup \{(b, \text{lb}, s)\})$$

- **RIGHTARC:** Assert a head-dependent relation between the word on the top of the stack and the word at the front of the input buffer; shift the word at the front of the input buffer to the stack.

$$\text{RIGHT}_{\text{lb}}[(\sigma|s, b|\beta, A)] = (\sigma|s|b, \beta, A \cup \{(s, \text{lb}, b)\})$$

- **SHIFT:** Remove the word from the front of the input buffer and push it onto the stack.
$$\text{SHIFT}[(\sigma, b|\beta,\ A)] = (\sigma|b, \beta, A)$$
- **REDUCE:** Pop the stack.
$$\text{REDUCE}[(\sigma|s, \beta,\ A)] = (\sigma, \beta, A)$$

Any configuration with an empty stack and a buffer containing only the ROOT is terminal.

In the Arc-Eager system, specific preconditions govern the legality of transition actions. For the RIGHTARC and SHIFT transitions, they are considered legal only when the first element of the buffer is not the ROOT. Additionally, the LEFTARC, RIGHTARC, and REDUCE transitions are legal only when the stack is non-empty. Furthermore, the LEFTARC transition is legal only when the top element on the stack (s_1) doesn't have a parent in the accumulated set A (the set of arcs in the parse tree). Conversely, the REDUCE transition is legal when s_1 does have a parent in A.

Step	Stack	Buffer	Action	Relation Added
0	[]	[book, me, the, morning, flight, ROOT]	SHIFT	
1	[book]	[me, the, morning, flight, ROOT]	RIGHTARC	(book→me)
2	[book, me]	[the, morning, flight, ROOT]	REDUCE	
3	[book]	[the, morning, flight, ROOT]	SHIFT	
4	[book, the]	[morning, flight, ROOT]	SHIFT	
5	[book, the, morning]	[flight, ROOT]	LEFTARC	(morning←flight)
6	[book, the]	[flight, ROOT]	LEFTARC	(the←flight)
7	[book]	[flight, ROOT]	RIGHTARC	(book→flight)
8	[book, flight]	[ROOT]	REDUCE	
9	[book]	[ROOT]	LEFTARC	(ROOT→book)
10	[]	[ROOT]	Done	

Table 3.2: Arc-Eager parsing example

The Arc-Eager system is different because its tree-building strategy is even more eager, adding arcs at the earliest possible moment while parsing. This eagerness ensures that the dependencies are formed quickly, hence facilitating efficiency in the parsing process. Importantly, each word collects all its left dependents before all its right ones in the Arc-Eager system. This ordering of dependencies generation shows a structural bias capturing the linguistic tendency for words to establish connections with their leftward dependents before their rightward ones. The combined approach of these preconditions and the eager construction makes the Arc-Eager system very valuable for the efficient and accurate parsing of natural language sentences.

Limitations of the Arc-Eager parser include potential challenges in handling non-projective sentences and certain linguistic phenomena that require a more flexible approach to attachment decisions. However, its strengths lie in its efficiency and ability to handle local dependencies well, making it suitable for certain parsing tasks where a more eager attachment strategy is advantageous.

3.2.3 Arc-Hybrid

The Arc-Hybrid parser is another approach in dependency parsing, offering a distinct set of rules to unravel the grammatical structures of sentences. This parsing model combines SHIFT, LEFTARC, and RIGHTARC actions, akin to the Arc-Standard parser, but introduces an additional set of rules to address specific linguistic phenomena. In the Arc-Hybrid parser, the initial configuration is the same as the Arc-Standard one; furthermore:

- **LEFTARC:** Assert a head-dependent relation between the word at the front of the input buffer and the word at the top of the stack, then pop the stack.
$$\text{LEFT}_{\text{lb}}[(\sigma|s, b|\beta, A)] = (\sigma, b|\beta, A \cup \{(b, \text{lb}, s)\})$$
- **RIGHTARC:** Establish a dependency arc between the second-to-top stack word and the top stack word, then remove the top element from the stack.
$$\text{RIGHT}_{\text{lb}}[(\sigma|s_1|s_0, \beta, A)] = (\sigma|s_1, \beta, A \cup \{(s_1, \text{lb}, s_0)\})$$
- **SHIFT:** Move a word from the buffer to the top of the stack.
$$\text{SHIFT}[(\sigma, b|\beta, A)] = (\sigma|b, \beta, A)$$

In the hybrid system, specific preconditions govern the legality of transition actions. The RIGHTARC transition is deemed legal only when the stack has at least two elements, ensuring a minimum structural context for the attachment of dependents. On the other hand, the LEFTARC transition is legal only when the stack is non-empty and the top element on the stack (s_1) isn't the ROOT, reflecting the requirement for a non-empty and non-ROOT configuration to allow left attachment.

Step	Stack	Buffer	Action	Relation Added
0	[ROOT]	[book, me, the, morning, flight]	SHIFT	
1	[ROOT, book]	[me, the, morning, flight]	SHIFT	
2	[ROOT, book, me]	[the, morning, flight]	RIGHTARC	(book→me)
3	[ROOT, book]	[the, morning, flight]	SHIFT	
4	[ROOT, book, the]	[morning, flight]	SHIFT	
5	[ROOT, book, the, morning]	[flight]	LEFTARC	(morning←flight)
6	[ROOT, book, the]	[flight]	LEFTARC	(the←flight)
7	[ROOT, book]	[flight]	SHIFT	
8	[ROOT, book, flight]	[]	RIGHTARC	(book→flight)
9	[ROOT, book]	[]	RIGHTARC	(ROOT→book)
10	[ROOT]	[]	Done	

Table 3.3: Arc-Hybrid parsing example

The hybrid system entails some actions of both the Arc-Standard system and the Arc-Eager system. It utilizes the LEFTARC action from the Arc-Eager system and the RIGHTARC action from the Arc-Standard system. This combination allows for a dynamic parsing process that benefits from the strengths of both strategies.

It builds trees bottom-up, just like the Arc-Standard system: every word has acquired all its dependents before being attached to its head. Additionally, the Arc-Hybrid parser, like the Arc-Eager system, must require all of its left dependents to be gathered before any right one. This combination of bottom-up construction and left-to-right dependency collection provides a balance between efficiency and accuracy, making the Arc-Hybrid parser a flexible and effective tool for parsing natural language sentences.

Even though its versatility, the Arc-Hybrid parser has its own limits. It works quite well in capturing a wide range of linguistic structures, but may face some problems where intricate syntactic or semantic subtlety demands more sophisticated parsing strategies. The dynamic balance between simplicity and expressiveness is a trade-off that influences the Arc-Hybrid parser's performance across different linguistic contexts.

On the other hand, there are a few strong points of the Arc-Hybrid parser. It is able to deal fairly well with a great number of syntactic constructions and can thus handle a broad variety of sentences. Its hybrid nature allows it to strike a balance between local and non-local dependencies, offering a pragmatic solution for applications where a compromise between computational efficiency and linguistic expressiveness is essential.

3.3 Oracle

The oracle is the focal point in the landscape of syntactic parsing: it provides the parser with the map through which the intricate web of grammatical relationships within the sentence can be navigated. In this regard, the oracle serves as a supervising body, defining the optimal sequence of transition actions that need to be taken so as to acquire a valid dependency tree. We consider the oracle to be a boolean function $o(t; c, T)$, the value of which is true if and only if the transition t is correct in configuration c with respect to the gold tree T.

It is important to note, however, that such a function could be defined in terms of many underlying functions that are also referred to as oracles. In other words, different criteria or sub-functions evaluating how the transition is correct in the given configuration can be given to construct the boolean oracle function. Those underlying functions will help to define the general oracle, and the choice of them may differ according to the parsing context or to the particular goals of the parsing system. This flexible definition of oracle provides its adaptation to diverse parsing models, algorithms, and objectives, offering a modular and customizable approach for defining the correctness of transitions during parsing.

The static oracle, the non-deterministic oracle, and the dynamic oracle are the three possible variants that we will focus on. The following paragraphs will unravel the features of each one of them to clarify how such oracles increase diversity and agility in the parsing process of NLP.

3.3.1 Static

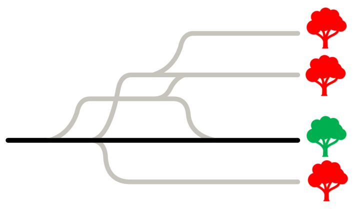


Figure 3.2: Static oracle's search space

The static oracle is the function $o_s(T)$ which takes a tree T to a sequence of transitions $t_1, ..., t_n$, guaranteeing that a predefined set of transition actions during the whole parsing process is followed. A static oracle is correct if starting from the initial configuration and applying transitions in $o_s(T)$ in order results in the transition system reaching a terminating configuration with gold parse tree T. All these definitely make the training easier and more feasible in creating the labeled data to train a machine learning model, nonetheless the static nature of the oracles also implies limitations in other respects. In parsing, static oracle translates to a limitation in the coverage and the completeness of the technique. Indeed, a static oracle is incomplete in the sense that it is defined only for such configurations that belong to the oracle path. In other words, it is functioning in a way where it either permits a single transition in a given configuration, or it is left undefined in that specific configuration.

This means that the static oracle might not offer advice or decisions for all the configurations that would be met while parsing. It focuses on a particular sequence of transitions, or path, and does not provide information or guidance on configurations that fall outside of it. For this reason, there might be configurations where the static oracle is silent or lacks a defined response.

Therefore, while static oracles are useful for some purposes, their basic limitation is that they are selective, in the sense that they cannot give guidance in general for all the possible configurations of the parser. It may have difficulty accommodating varied sentence structures, sometimes with parsing results of lesser quality. Specifically, it is inflexible in the face of parsing challenges that are typically non-local in nature and have complicated syntactic formations. However, within these constraints, the static oracle is very appropriate for situations where consistency and simplicity in training data generation trump the finer need of nuanced adaptation. The strengths of this approach are the ease of implementation and the ability to produce consistent training sets, facilitating a parser that emphasizes stability over dynamic, responsive qualities.

3.3.2 Non-Deterministic

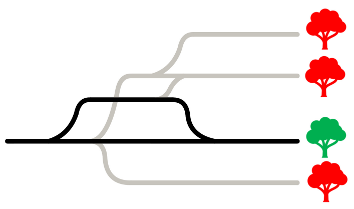


Figure 3.3: Non-deterministic oracle's search space

The non-deterministic oracle brings in it a kind of flexibility to the parsing process, in the sense that at every point of the parse, a number of valid transition actions can take place. On the other hand, the static oracle only admits of the fact that there is but one path that can possibly lead to a correct syntactic structure for a given sentence. Hence, a non-deterministic oracle is a function $o_n(c, T)$ that maps a configuration c and a tree T to a set of transitions. Such an oracle is correct if, and only if, for each projective dependency tree T, for each configuration c from which T is reachable, and for every transition $t \in o_n(c, T)$, $t(c)$ is a configuration from which T is still reachable. However, this definition of correctness for non-deterministic oracles is restricted to configurations from which the gold tree is reachable. Non-deterministic oracles offer more flexibility than static ones by allowing spurious ambiguity, meaning they support the possibility of different sequences of transitions leading to the gold tree. Nevertheless, it's essential to note that they are still only guaranteed to be correct on a subset of possible configurations: if T is not reachable from c, $o_n(c, T)$ is not necessarily well-defined.
The non-deterministic oracle adaptability proves beneficial in capturing the inherent ambiguity and variability present in natural language. However, in spite of their flexibility, they also represent certain difficulty in the training process: the ambiguity brought about by numerous valid actions calls for its careful treatment in order to guarantee its effective learning and generalization. Also, parsing could become complicated since the process involves the parser navigating a tree-like decision space. Even though non-deterministic oracles could be somewhat hard to handle, they are versatile in terms of sentence structures and linguistic phenomena. This makes them a valuable choice in scenarios of flexibility and of the ability to handle varying syntactic constructions become crucial. The nuanced approach of non-deterministic oracles reflects an understanding of essential intricacies and the multifaceted nature of language parsing. Such flexibility facilitates more realistic approaches to the ambiguities of natural languages, enhancing the robustness of the parsing model and making it applicable in real-life situations.

3.3.3 Dynamic

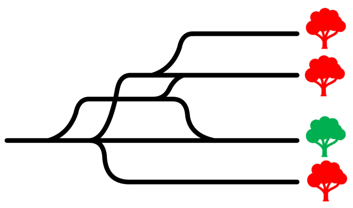


Figure 3.4: Dynamic oracle's search space

The dynamic oracle represents a more sophisticated version in that the guiding responses are adapted dynamically with the changing state in the parsing process. Unlike both the static and non-deterministic oracles, the dynamic oracle considers the parser's current configuration, allowing it to provide contextually tailored guidance. In other words, a dynamic oracle is a function $o_d(c, T)$ that, given the parser's current configuration c, computes the cost for each possible transition in terms of gold dependency arcs lost with respect to the gold dependency tree T.

Following Goldberg and Nivre (2013)[6]:

- the cost function $C(A, T)$ measures, through the Hamming loss, the cost of outputting parse A when the gold tree is T;
- the cost function $C(t; c, T)$ for transition t at configuration c is defined as the difference in cost between the best tree reachable from $t(c)$ and c, respectively, that is:

$$C(t; c, T) = \min_{A:t(c)\to A} C(A, T) - \min_{A:c\to A} C(A, T)$$

The dynamic oracle returns the set of transitions with zero cost:

$$o_d(c, T) = \{t | C(t; c, T) = 0\}.$$

As long as correctness is concerned, a dynamic oracle is deemed correct if and only if, for every projective dependency tree T with yield W, every reachable configuration c from the initial configuration $\textsc{initial}_{(W)}$, and every transition $t \in o_d(c, T)$, the minimum cost of the parser reachable from c equals the minimum cost of the parser reachable from $t(c)$. This holds even if the gold tree T is no longer directly reachable. More formally:

$$\min_{A:c\to A} C(A, T) = \min_{A:t(c)\to A} C(A, T).$$

Additionally, it is also possible to extend the dynamic oracle to non-projective examples returning the set of minimum-cost transitions, as explored by Aufrant et al. (2018) [2]:

$$o_d(c, T) = \{t \mid \min_t C(t; c, T)\}.$$

In conclusion, the dynamic oracle adaptability is valuable in the complex sentence structure handling, effectively resolving possible ambiguities. The power of the dynamic oracle consists of capturing non-local dependencies and intricate syntactic relationships, which is beneficial for parsing tasks where a set of transition actions may fall short. On the other hand, the dynamic oracle's complexity of implementation and continuous real-time adjustments pose challenges during training. The parser has to learn to navigate a decision space that undergoes dynamic change and, hence, needs a more complicated learning process. Overall, it would seem that these problems aside, the dynamic oracle is remarkable in that it possesses the ability of capturing the granular subtleties of natural language syntax to afford a responsive and context aware approach to syntactic parsing.

3.3.4 Summary

Transition-based parsers are known for their speed, but often face challenges related to error propagation. This issue is exacerbated by their typical training using deterministic and incomplete oracles (static/non-deterministic). These oracles assume a unique canonical path throughout the transition process and are only valid as long as the parser adheres to this specific path.

By exploiting a dynamic oracle during training, the parser gains the ability to explore alternative and non-optimal paths. This adaptability is particularly useful in resolving complex situations, such as handling non-projectiveness. Dynamic oracles provide a more flexible training approach, allowing the parser to learn from a broader range of scenarios and improving its ability to handle linguistic phenomena that may deviate from the canonical path assumed by deterministic oracles.

In the next chapter we will explain the fundamental property that a transition system is required to possess in order to guarantee the derivation of an exact dynamic oracle with linear complexity.

4 Arc-Decomposition Property

This chapter explores a crucial concept in dependency parsing, focusing on how the arc decomposition property simplifies parsing by reducing the reachability of arc sets or trees to individual arcs, aiding in the development of dynamic oracles. Arc reachability, set reachability, tree consistency, and arc decomposition are defined and examined in the context of different parsers. It's shown that while the Arc-Eager and Arc-Hybrid parsers possess the arc decomposition property, allowing them to efficiently parse tree-consistent arc sets, the Arc-Standard parser lacks this property. This limitation in the Arc-Standard parser is due to the potential for projective trees consistent with arcs where both nodes are still on the stack, impacting its ability to derive certain tree structures. Thus, the chapter underscores the varying capabilities and limitations of different parsing systems in handling complex syntactic structures.

Arc decomposition is a powerful property, allowing us to reduce reasoning about the reachability of arc sets or trees to reasoning about the reachability of individual arcs, and can be used to derive dynamic oracles for the Arc-Eager and Arc-Hybrid systems. Firstly, let's introduce some formal definitions:

- **Arc Reachability:** We say that a dependency arc (h, d) is reachable from a configuration c, written $c \rightarrow (h, d)$, if there is (possibly empty) sequence of transitions $t_1, ..., t_k$ such that $(h, d) \in A_{(t_k(...t_1(c)))}$. In words, we require a sequence of transitions starting from c and leading to a configuration whose arc set contains (h, d).
- **Arc Set Reachability:** A set of dependency arcs $A = \{(h_1, d_1), ..., (h_n, d_n)\}$ is reachable from a configuration c, written $c \rightarrow A$, if there is a (possi-

bly empty) sequence of transitions $t_1, ..., t_k$ such that $A \subseteq A_{(t_k(...t_1(c)))}$. In words, there is a sequence of transitions starting from c and leading to a configuration where all arcs in A have been derived.

- **Tree Consistency:** A set of arcs A is said to be tree consistent if there exists a projective dependency tree T such that $A \subseteq T$.
- **Arc Decomposition:** A transition system is said to be arc decomposable if, for every tree consistent arc set A and configuration c, $c \longrightarrow A$ is entailed by $c \longrightarrow (h, d)$ for every arc $(h, d) \in A$. In words, if every arc in a tree consistent arc set is reachable from a configuration, then the entire arc set is also reachable from that configuration.

Let's now prove the arc decomposition property for the parsers we previously described.

4.1 Arc-Eager parser

For the Arc-Eager system, consider an arbitrary configuration $c = (\sigma, \beta, A)$ and a tree-consistent arc set A' such that all arcs are reachable from c. We can partition A' into four sets, each of which is by necessity itself a tree-consistent arc-set:

- $\overline{B} = \{(h, d) | h, d \notin \beta\}$
- $B = \{(h, d) | h, d \in \beta\}$
- $B_h = \{(h, d) | h \in \beta, d \in \sigma\}$
- $B_d = \{(h, d) | d \in \beta, h \in \sigma\}$

Arcs in $\overline{B}$ are already in A and cannot interfere with other arcs. B is reachable by any sequence of transitions that derives a tree consistent with B for a sentence containing only the words in β. In deriving this tree, every node x involved in some arc in B_h or B_d must at least once be at the head of the buffer. Let c_x be the first such configuration. From c_x, every arc $(x, d) \in B_h$ can be derived without interfering with arcs in A' by a sequence of REDUCE and LEFTARC$_{lb}$ transitions. This sequence of transitions will trivially not interfere with other arcs in B_h. Moreover, it will not interfere with arcs in B_d because A' is tree consistent and projectivity ensures that an arc of the form (y, z) $(y \in \sigma, z \in \beta)$ must satisfy $y < d < x \leq z$. Finally, it will not interfere with arcs in B because the buffer remains unchanged. After deriving every arc $(x, d) \in B_h$, we remain with at most one $(h, x) \in B_d$ (because of the single-head constraint). By the same reasoning as above, a sequence of REDUCE and LEFTARC$_{lb}$ transitions will take us to a configuration where h is on top of the stack without interfering with arcs in A'. We can then derive the arc (h, x) using RIGHTARC$_{lb}$. This does not interfere with arcs remaining in B_h or B_d because all such arcs must have their buffer node further down the buffer (due to projectivity). At this point, we have reached a configuration c_{x+1} to which the same reasoning applies for the next node $x + 1$.

4.2 Arc-Hybrid parser

The proof for the hybrid system is very similar but with a slightly different partitioning because of the bottom-up order and the different way of handling right-arcs.

4.3 Arc-Standard parser

The arc decomposition property, which holds for the Arc-Eager system, does not apply to the Arc-Standard system. To illustrate this, consider a configuration with the stack $\sigma = a, b, c$. While the arc (c, b) is reachable via LEFTARC, and the arc (b, a) is reachable via RIGHTARC followed by LEFTARC, the arc set $A = \{(c, b), (b, a)\}$ forms a projective tree and is thus tree consistent. However, it is easy to see that A is not reachable from this configuration.

The reason for the failure of the proof technique in the Arc-Standard system lies in the fact that the arc set corresponding to B in the Arc-Eager system may involve arcs where both nodes are still on the stack. Consequently, there is no guarantee that all projective trees consistent with these arcs can be derived. In a very similar hybrid system, such arcs exist as well, but they are limited to arcs of the form (h, d) where $h < d$, and h and d are adjacent on the stack. This restriction is sufficient to restore arc decomposition.

5 Dynamic Oracle: Exact *vs.* Approximate

This chapter will discuss details of how the Arc-Standard parser works, particularly in relation to the notion of dynamic oracles, which has a major part in raising the performance level of the parser, due to the ability in handling the natural complexity of non-projective dependency structures. The discussion navigates through the nuances of exact and approximate dynamic oracles, shedding light on their operational mechanisms, computational implications, and their critical function in the context of transition-based dependency parsing. Starting from the previous Chapter 4, where we dissected the arc-decomposition property and its effects on the feasibility and efficiency of dynamic oracles, we offer now a thorough analysis of the trade-offs between computational complexity and parsing accuracy. The discussion provides details of two distinct dynamic oracles, exact and approximate, each carefully tuned to optimize the performance of the Arc-Standard parser, while conscientiously navigating the challenges posed by non-projective dependency structures. Through this exploration, Chapter 5 aims to present, in detail, the strategies used for enhancing the ability of the Arc-Standard parser to handle intricate linguistic constructs in a more efficient and effective manner.

As noted in Chapter 3, transition-based dependency parsers have partial effectiveness when dealing with non-projective dependency structures due to certain limitations they encounter in crossing arc-processing. For the identified problem, solutions can be found from the use of dynamic oracles, as shown in Chapter 2, making these parsers better in terms of performance, generally maintaining linear time complexity.

However, an exact dynamic oracle with a linear runtime complexity for every parser is not always possible. Chapter 4 states that for such a derivation to be easily implemented, the transition system must have the arc decomposition property, which "decomposes" reachability reasoning on arc sets or trees into reachability reasoning of single arcs.

It still makes sense to use the dynamic oracle even in the case of the Arc-Standard, but providing exactness comes at a very heavy cost with regard to time complexity.

To delve into the concept of a dynamic oracle, the main objective aims at the determination of an action cost computing function able to evaluate the loss concerning gold arcs caused by potential parser transitions. In the context of the Arc-Standard parser, these transitions are LEFTARC, RIGHTARC, and SHIFT. This computational capability, thus, equips the model with the ability to process non-projective sentences during training efficiently, avoiding not only a wasteful discard of data, but also enhancing the model's robustness in scenarios where there is no unequivocally correct answer. In such cases, the model may decide to choose the "lesser evil", throwing away some gold arcs in order to save more, generating a parsing tree that is as close as possible to the gold dependency tree.

The following chapter describes two dynamic oracles for the Arc-Standard-parser: an exact and an approximate one. Both oracles try to solve the challenges posed by non-projective dependency structures, taking under consideration the balance between accuracy and computational efficiency.

5.1 EXACT DYNAMIC ORACLE

This oracle, initially introduced in an unpublished draft paper, is designed to enhance the Arc-Standard parser. While the paper itself wasn't published, we'll outline its key concepts below.

The fundamental idea behind the dynamic oracle for the Arc-Standard parser involves the computation of the quantity $L(c, t_G)$. This quantity represents the minimum loss concerning t_G, signifying the smallest loss associated with a tree that the Arc-Standard parser can generate when parsing from a given configuration c. By calculating this quantity both before and after a transition τ, it becomes possible to precisely determine the cost incurred by this transition. This particular approach is especially effective in handling non-projective trees, showcasing the parser's ability to navigate and parse complex linguistic structures accurately.

To facilitate the explanation, let's introduce some auxiliary notation. Given a string w, a gold dependency tree t_G, and a configuration $c = (\sigma, \beta, A)$ obtained by parsing w, let $\gamma = \sigma\beta$ be the concatenation of the stack and the buffer. We denote $\gamma[i]$, $0 \leq i < |\gamma|$, as the i-th element of γ, and use $b = |\sigma|$ to indicate the boundary between σ and β in γ, such that $\gamma[b-1] = \sigma[0]$ and $\gamma[b] = \beta[0]$.

An $|\gamma| \times |\gamma|$ array T is introduced, where each element $T[i, j]$ is an association list with key h in V_w (the vocabulary of words in w) such that $i \leq h \leq j$. The entry $T[i, j](h)$ stores the minimum loss with respect to t_G of a dependency tree with head h, obtained through a reduction of the nodes $\gamma[i], ..., \gamma[j]$. Notably, $T[0, |\gamma| - 1](0)$ is the desired quantity $L(c, t_G)$.

Algorithm 1 employs dynamic programming to fill in the association lists in T. Specifically, each loss is computed as a function of the loss of the two trees being combined in a reduction and the contribution of the new arc created in the reduction itself. This dynamic programming approach enables the efficient calculation of the minimum loss for various dependency trees during the parsing process.

Algorithm 1 Loss function computation for Arc-Standard exact dynamic oracle

```
1:  for $i \leftarrow 0$ to $b - 1$ do
2:      $T[i, i](i) \leftarrow L(\gamma[i], t_G)$
3:  end for
4:  for $i \leftarrow b$ to $(|\gamma| - 1)$ do
5:      $T[i, i](i) \leftarrow 0$
6:  end for
7:  for $j \leftarrow (b - 1)$ to $(|\gamma| - 1)$ do
8:      for $i \leftarrow j$ downto 0 do
9:          FillTableEntry($T$,$i$,$j$)
10:     end for
11: end for
12: return $T[0, |\gamma| - 1](0)$
```

Algorithm 1 can be summarized as follows:

- Initialization (lines 1-6): in the initialization phase, the algorithm begins by setting up the main diagonal of the matrix. It distinguishes between elements originating from the stack and those from the buffer. If an element of γ comes from the stack, its loss contribution may be non-zero, reflecting the impact of previous computations resulting in c. Conversely, if the element comes from the buffer, its loss contribution is consistently 0.

- Filling upper portion of T (lines 7-11): moving on to the next phase, the algorithm focuses on populating the upper portion of matrix T. This process involves systematically processing columns from left to right and visiting each column from bottom to top. Notably, the algorithm skips entries $T[i, j]$ with $i < b - 1$ during this step: these entries correspond to reductions of trees situated below the top-most tree in the stack. Such reductions are deemed invalid in the context of Arc-Standard computations and are, therefore, excluded from consideration (these entries represent reductions of trees below the top-most tree in the stack, which arent allowed).

Algorithm 2 Filling Table Procedure

```
1:  procedure FillTableEntry($T$,$i$,$j$)
2:  for $k \leftarrow i$ to $(j-1)$ do
3:      for each key $h_l$ defined in $T[i,k]$ do
4:          for each key $h_r$ defined in $T[k+1,j]$ do
5:              $\text{loss}_{\text{la}} \leftarrow T[i,k](h_l) + T[k+1,j](h_r) + \delta_G(h_l \leftarrow h_r)$
6:              $T[i,j](h_r) \leftarrow \min\{\text{loss}_{\text{la}}, T[i,j](h_r)\}$
7:              $\text{loss}_{\text{ra}} \leftarrow T[i,k](h_l) + T[k+1,j](h_r) + \delta_G(h_l \rightarrow h_r)$
8:              $T[i,j](h_l) \leftarrow \min\{\text{loss}_{\text{ra}}, T_{[i,j](h_l)}\}$
9:          end for
10:     end for
11: end for
```

This procedure systematically determines all valid combinations for reducing a tree that spans $\gamma[i], ..., \gamma[k]$ with root h_l and another tree spanning $\gamma[k+1], ..., \gamma[j]$ with root h_r. Each identified combination yields a composite tree t that spans $\gamma[i], ..., \gamma[j]$. The root of t is h_r if the reduction involves a transition τ = LEFTARC (as indicated in lines 5 and 6), and h_l if τ = RIGHTARC is employed (as denoted in lines 7 and 8). The loss associated with t is calculated as the sum of the losses of the two trees being combined, augmented by a function δ_G pertaining to the arc created by τ. This function is defined by:

$$\delta_G = \begin{cases} 0, & \text{if } (i \rightarrow j) \text{ is an existing arc in } t_G \\ 1, & \text{otherwise} \end{cases}$$

Here, δ_G takes on a value of 0 if the arc $(i \rightarrow j)$ is already present in the tree t_G, and 1 otherwise.

5.2 Approximate Dynamic Oracle

Algorithm 1 runs with space complexity of $\mathcal{O}(|\gamma|^3)$ and time complexity of $\mathcal{O}(|\gamma|^5)$, where γ denotes the concatenation of the stack σ and the buffer β. In practice, although the stack is very much smaller than the input string w, at the start of the computation, the buffer β is of the same order of magnitude as w, and this initial condition has a significant effect on the computational cost of the algorithm.

Despite the potential for optimization by precomputing independent reductions in β and making use of the "split" technique introduced by Eisner and Satta (1999) [4], resulting in a shortened buffer β_{red} and achieving an improved asymptotic running time of $\mathcal{O}(|\sigma\beta_{red}|^3)$, this enhancement has not been implemented in our approach. Instead, we directly pursued a linear approximation.

To address this computational challenge, the approximation technique leverages the identification of distinct configurations within the algorithm. Subsequently, a more lenient cost function is extrapolated, allowing for an approximation of the cost associated with each transition. The effectiveness of this approximation strategy becomes evident when comparing it to the exact method, particularly when scrutinizing the set of transitions with minimum cost. Remarkably, despite its linear runtime, the approximation closely aligns with the results obtained through the exact approach.

It is essential to note that while the approximation was originally devised for projective sentences, experimental results indicate its effectiveness extends to non-projective sentences as well.

5.2.1 Leftarc and Rightarc

Concerning the arc-generating transitions, namely leftarc and rightarc, their precise cost can be efficiently computed in linear time. To begin, an illegal transition is flagged with an infinite cost. Such illegality occurs under the following conditions:

- If there are fewer than two elements in the stack, as two tokens are required to generate an arc.

- If there are exactly two elements in the stack but the buffer isn't empty. This is because the ROOT token is invariably placed at the beginning of the sentence, and the last arc to be generated is consistently the one between the ROOT and the last remaining token of the sentence, irrespective of the correctness of the arc.
- In the case of a LEFTARC transition, it is also deemed illegal when there are two elements in the stack and none in the buffer. This is because the ROOT node cannot serve as a dependent.

After checking the legality of the transition, in either cases the dependent token (i.e. the top stack element for RIGHTARC, the second top stack element for LEFTARC) is the one to be eliminated from the stack, meaning that it won't be possible to create other arcs between the dependent and any other token in either the stack or the buffer. Therefore, in the unfortunate case one or more child of the dependent haven't been attached to it, the parser won't be able to retrieve those gold arcs, resulting in a loss. By scanning both the stack and the buffer, we can count the number of gold arcs involving the soon to be eliminated dependent: this is the cost of a legal LEFT/RIGHT-ARC transition.

In Figure 5.1 both LEFTARC and RIGHTARC are legal, so we examine $w3$ for LEFTARC and $w4$ for RIGHTARC. As long as the former is concerned, there is still a gold arc involving it, (i.e. $(w2, w3)$), that won't be possible to generate if the LEFTARC transition is performed; however this loss isn't actually due to this specific transition, but rather to a previous one (i.e. a wrong SHIFT), therefore $\text{cost}_{\text{LEFTARC}} = 0$[1]. Performing a RIGHTARC, on the other hand, will results in a $\text{cost}_{\text{RIGHTARC}} = 4$, given that the gold arcs $[(\text{ROOT}, w4), (w4, w2), (w4, w5), (w4, w7)]$ will be impossible to generate.

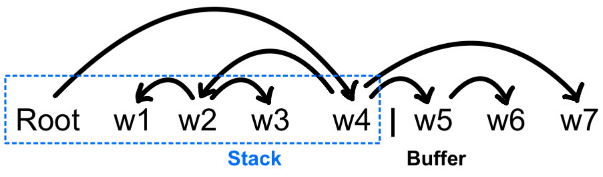


Figure 5.1: LEFT/RIGHT-ARC cost computation

5.2.2 SHIFT

Let's shift our focus to the third transition for the moment, deferring the completion of the cost computation for the LEFT/RIGHT-ARC moves until later, as it is tied to the SHIFT operation.

Once again, we assess the legality of the transition, noting that a SHIFT operation incurs an infinite cost if the buffer is empty. Subsequently, we examine the top stack element s_1, where three distinct cases may unfold.

CASE 1 If the top stack element s_1 has a right child, the SHIFT on s_1 does not introduce any loss in terms of gold arcs: we can collect its left children first and then the right ones, or vice versa, without compromising any gold arcs. In the configurations represented by Figure 5.2 both LEFTARC and SHIFT have cost = 0.

In configuration 5.2a, $s_1 = w2$: given that $w1$ is a left child of $w2$ and $w3$ is a right one, we can decide to immediately collect $w1$ or do that after shifting $w3$ in the stack and attaching it to its parent through RIGHTARC. Therefore $\text{cost}_{\text{SHIFT}} = 0$, since no gold arc in the future will be lost because of a SHIFT transition in the current configuration.

In configuration 5.2b, $s_1 = w4$: once again we have $\text{cost}_{\text{LEFTARC}} = 0$[1] and $\text{cost}_{\text{SHIFT}} = 0$, while $\text{cost}_{\text{RIGHTARC}} = 4$. Focusing on SHIFT, this transition doesn't increment the number of gold arcs that will inevitably be lost, as the same configuration can be reached again after processing the right children of $w4$ without compromising the gold arcs currently in the stack. Evidently there is no guarantee that $w4$'s right children will be correctly attached, however possible errors cannot be traced back to a SHIFT in the current configuration, but have to be assigned to future wrong moves: every reachable gold arc in the current configuration is still reachable after a SHIFT transition.

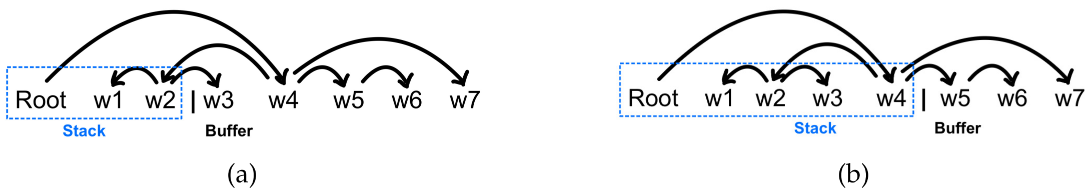


Figure 5.2: SHIFT cost computation CASE 1

CASE 2 If the top stack element s_1 lacks a right child and s_1 is itself a right child, a comparison between s_1 and the first buffer element b_1 is necessary.

[1]LEFTARC does lose the arc ($w2$, $w3$) generating ($w4$, $w3$), however the gold arc was already lost due to a previous erroneous SHIFT: therefore the actual cost computed by the approximation is 0. Further details on this matter are provided at the conclusion of this section.

- CASE 2.1: If at any point in the "lineage" (i.e. set of forefathers including the examined token) of b_1 there's an "orphan" (i.e. a token not yet assigned as a dependent and whose gold parent has already been eliminated from the stack), excluding the ROOT token, we return a cost of 0. Once again, there is no guarantee that no additional errors will be made. However, it is possible to generate the arc (s_1, orphan), resulting in the best case scenario: where the same current configuration is reached after having correctly processed the sub-tree whose root is the orphan token.

  Considering Figure 5.3 below, $s_1 = w6$, $b_1 = w7$: in this configuration a previous wrong move generated the arc ($w3$, $w4$), losing all the dependencies marked with a red X and removing $w4$ from the stack. Since b_1 is an orphan and attaching it to s_1 doesn't lose additional arcs, SHIFT is costless.

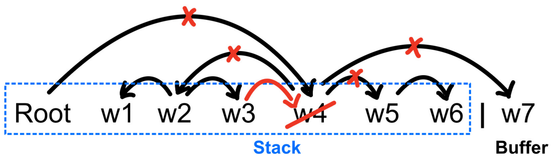


Figure 5.3: SHIFT cost computation CASE 2.1

- CASE 2.2: If at any point in the "lineage" of b_1 one of the forefathers is in the stack (excluding the ROOT) we return a cost of 1. We rely on the LEFT/RIGHT-ARC cost computations to determine the optimal move, likely a RIGHTARC, sacrificing the gold arc between the forefather in the stack and its dependent in the buffer. There is no guarantee that the parser will actually sacrifice that specific gold arc, however a SHIFT in the current configuration will make us loose at least that one gold dependency: we can charge this SHIFT transition of that loss only, while any additional one is caused by further erroneous moves.
  In Figure 5.4 below, $s_1 = w6$ and $b_1 = w7$: b_1's father (i.e. $w4$) is currently present in the stack, and it's not the ROOT. If we choose to shift b_1 into the stack, it will lead to a configuration where the most favorable outcome is sacrificing the existing arc between b_1 and its father (in this case $w4$), attaching b_1 to a different token. However, it is still possible to correctly attach any potential children of b_1 to it. Consequently, to avoid further losses, the only gold arc that is required to be lost in this scenario is the one between b_1 and its father, resulting in a cost of 1. This same reasoning can be applied to any of b_1's forefathers.

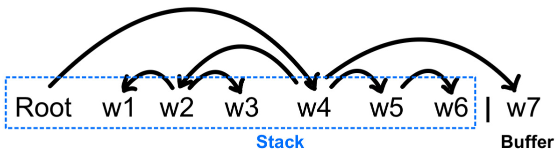


Figure 5.4: SHIFT cost computation CASE 2.2

- CASE 2.3: If neither of the previous conditions is met, we approximate the cost of a SHIFT as the minimum number of gold arcs between those involving the top stack element s_1 and those concerning the first buffer element b_1. This scenario arises when b_1 is the dependent of the ROOT token, thus it is highly probable that the minimum cost will correspond to the number of gold arcs that involve the token s_1[2].

Considering Figure 5.5, where $s_1 = w3$ and $b_1 = w4$, the returned $\text{cost}_{\text{SHIFT}} = \min\{1, 4\}$. It's easy to notice that, once w_4 is in the stack, the best case scenario will consist in attaching $w1$, $w2$ and $w3$ to $w4$, resulting in an actual loss of 2 gold arcs. Once again, in order to maintain a low computational complexity we decided to consider only the gold arcs involving s_1, but a more accurate approximation could be reached by examining the dependencies concerning s_1's brothers (in this case $w1$).

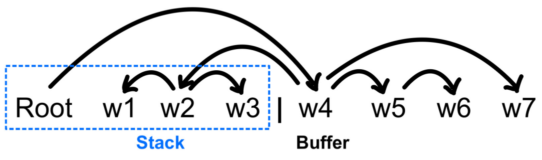
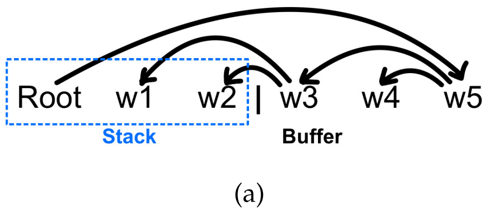


Figure 5.5: SHIFT cost computation CASE 2.3

CASE 3 If the top stack element s_1 lacks a right child and it is a left child, the number of its remaining children (all in the stack following the initial projectiveness hypothesis of our approximation) roughly represents the number of gold arcs that will be lost. Let's consider Figure 5.6a: $s_1 = w2$ and it has no right nor left child, therefore the SHIFT transition is costless. On the other hand, in 5.6b, where $s_1 = w3$, the approximate oracle calculates a returned cost of 2 since both $w1$ and $w2$, children of s_1, are still in the stack. However, the actual cost should be 1: this is because the optimal strategy for the parser is to sacrifice the arc ($w5$, $w4$) in favor of creating the wrong dependency ($w3$, $w4$). To obtain the exact cost, additional analysis and computation would be required, which could increase the complexity of our algorithm. Therefore, to balance accuracy and efficiency, we have opted for a more punitive but quicker approximation.

[2]Usually the ROOT's dependent (which in the described case is b_1) is the most important word in the sentence: the one with most dependents, therefore the one involved in more gold arcs. Thus, it is highly probable that the minimum number of gold arcs between those involving the ROOT's dependent and those concerning any other token of the sentence corresponds to the latter set.



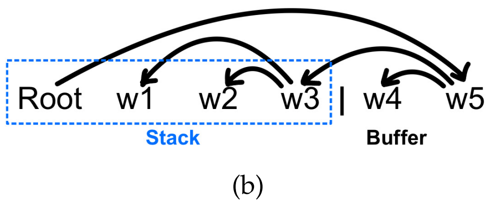


Figure 5.6: SHIFT cost computation CASE 3

The challenge of devising a linear exact dynamic oracle for the Arc-Standard parser stems from the nature of the SHIFT transition. Unlike the immediate loss in terms of gold arcs caused by the generation of a wrong arc, the repercussions of a wrong SHIFT are deferred and incurred later in the parsing process.

It is accurate to state that the gold arcs, which can no longer be generated due to a wrong SHIFT, do not register as losses until a subsequent LEFT/RIGHT-ARC transition materializes, creating incorrect arcs and resulting in an actual loss in terms of gold arcs. However, it's crucial to recognize that the responsibility for these lost gold arcs doesn't lie with the last LEFT/RIGHT-ARC move; instead, they were destined to be lost due to a preceding wrong SHIFT transition. The key lies in precisely computing the number of these gold arcs and distinguishing losses attributable to a SHIFT from those caused by an arc-generating move.

To address this complexity, the costs of one or more prior wrong SHIFT transitions must be retained. This way, when computing the cost of a LEFT/RIGHT-ARC, each move is penalized solely for its own "wrongness", without being unfairly burdened by the delayed consequences of something that occurred earlier. Furthermore, distinctions must be made among wrong SHIFTS. Given our focus on an approximation, we consider only the preceding move: if it is a SHIFT, the costs of subsequent LEFT/RIGHT-ARCS will be discounted by the cost of that previous SHIFT.

6 Results: part I

Chapter 6 discusses in detail the outcome obtained with the implementation of dynamic oracles in the domain of dependency parsing, particularly with the Arc-Standard parser. Central to our investigation is the utilization of the Universal Dependencies (UD) dataset, a comprehensive and standardized collection of annotated treebanks across many languages that has been central in driving forward research within NLP. This dataset's structured and uniform annotation schema allows us to model complex syntactic relations and effectively train dependency parsers.

We investigate the intricacies of unlabeled dependency parsing by construction of gold dependency trees using only the 'head' attribute for each data entry. This approach necessitates the inclusion of a dummy ROOT to anchor these trees, underscoring the root node's significant impact on parsing outcomes as identified in prior research. The large diversity of languages in the UD dataset and its associated standard annotations provides a very strong basis for our analysis, making it possible to evaluate the parser's performance across different linguistic settings.

We devise a methodological framework using the BiLSTM architecture, augmented by a Multi-Layer Perceptron (MLP), to distill rich feature representation at the word level and facilitate imitation learning. This setup, which avoids complex feature engineering in favor of the LSTM's native capabilities, follows the recent academic literature around dependency parsing. Despite the possibility to exploit more advanced models like BERT, our choice is justified by a balance between training efficiency and performance metrics.

Through a rigorous evaluation process, we assess the accuracy of our approximate dynamic oracle against its exact counterpart, revealing a close alignment in identifying minimum cost actions. This analysis not only underscores the computational efficiency of our approximation, but also highlights its effectiveness in enhancing parser performance, particularly in handling non-projective sentence structures.

As we dissect the performance metrics, measured by the Unlabeled Attachment Score (UAS), across various oracle implementations, we unveil the nuanced benefits of dynamic oracles in parsing. The data suggests a pronounced advantage in datasets with a higher incidence of non-projective sentences, illustrating the dynamic oracle's capacity to harness the full spectrum of training data more effectively than static oracles.

This chapter, therefore, not only presents a thorough examination of the results stemming from the application of dynamic oracles in dependency parsing, but it also offers insights into the potential for further advancements in parser performance, especially in the context of non-projective sentence parsing.

6.1 Dataset

Universal Dependencies (UD) dataset available on Hugging Face is a crucial resource for dependency parsing in the field of NLP. This dataset is an extensive collection of annotated treebanks across multiple languages, designed to foster shared practices and enhance NLP research globally. The structure of the UD dataset is meticulous and standardized, enabling it to capture complex syntactic relationships within sentences.

Each entry in the UD dataset includes a sentence from a particular language along with its linguistic annotations. These annotations follow a consistent scheme regardless of the language, which is one of the key strengths of this dataset. The annotations cover aspects like the part of speech (POS) for each word, morphological features, and crucially, dependency relations between words. These relations are annotated in a tree structure, where each word is linked to its head (or parent) in the sentence, along with a label describing the type of syntactic relation between them.

Here there is an entry example:

idx	text	tokens	lemmas	upos	xpos	feats	head	deprel	deps	misc
en_lines-ud-train-doc1-1	Show All	["Show", "All"]	["show", "all"]	[16, 11]	["IMP", "TOT-PL"]	["{'Mood': 'Imp', 'VerbForm': 'Fin'}", "{'Case': 'Nom'}"]	["0", "1"]	["ROOT", "obj"]	["None", "None"]	["None", "None"]

Table 6.1: Dataset entry example

In our studies, the unlabeled dependency parsing we aimed at only needs to construct the gold standard dependency tree with the 'head' attribute of every entry. The head of the ROOT dummy node is simply added at the beginning of the 'head' array. We use '-1' for this but any negative integer can be used. The ROOT's position is pivotal, significantly affecting the parser's efficacy; the variation of the position of the ROOT node has been studied in the work of M. Ballesteros et al. (2013) [3], where the diversity in parsing performance is generally related to it.

The significance of the UD dataset lies in its universality and diversity. By providing a unified annotation schema across a wide range of languages, it empowers researchers and developers in the development of more robust models and tools that can be transferred across a plurality of languages without much difficulty. This universality is particularly valuable for training advanced dependency parsers, which are essential tools in many NLP applications like machine translation, information extraction, and text summarization.

Furthermore, the UD dataset is ever-growing, thanks to thousands of community contributions from the global linguists and computer scientists. This not only helps to improve the quality and size of the dataset, but makes it stay contemporary and meaningful for the world of modern NLP.

Nowadays, this means availability for over 100 languages in terms of data. This high level of multilingual coverage is one of the best and makes it a particularly valuable resource not only for dependency parsing generally, but also for other NLP tasks within such very diverse linguistic contexts. Standard annotations in so many languages, supported by the UD dataset, make the creation and evaluation of of models that are both linguistically inclusive and robust.

Here there is a brief analysis of the languages we used for our experiments:

Dataset	% Non-Projective	# sentences	average sentence length	# tokens	# types	weighted average arc span[1]
en_lines	8.07	5243	17.97	94217	11096	2.4
ko_kaist	21.7	27363	12.79	350090	97870	2.0
grc_proiel	37.52	17080	12.53	213999	32983	2.1
grc_perseus	63.87	13919	14.58	202989	41562	2.8

Table 6.2: Dataset analysis

6.2 Accuracy

We measure the accuracy of our approximation against the exact dynamic oracle across the training datasets of different languages. For every sentence in the training data, the cost of every possible transition in every configuration in a pseudo-random path was computed using both oracles. We identified the set of transitions with the lowest cost according to each oracle and measured how much of the set from the approximate oracle was contained in the set from the exact oracle. Additionally, using the Spearman's rank correlation coefficient, we measured the similarity between the rankings of transitions; these rankings were based on the costs provided by the two oracles.

The Spearman's rank correlation coefficient (ρ) is a statistical measurement that evaluates the monotonic relation between two ranked variables. It tells how well the relation between these variables can be explained by a consistent increase or decrease in ranks. Being a measure on a scale between -1 and 1, the coefficient is particularly useful for non-linear relationships and analysis with ordinal data. A perfect monotonic relationship is indicated by a value of +1 or -1; 0 is an indication of the lack of such a relationship. The Spearman's ρ is advantageous since it has the ability to incorporate non-linear and ordinal data, and it is also robust to outliers since it has its computation based on ranks rather than real values. In assessing approximate and exact oracles within NLP, Spearman's ρ is applied to compare the ranking of transitions based on their costs, since this informs on how similarly the oracles prioritize decisions, thus offering a concise evaluation of an approximate oracle's effectiveness in mimicking the exact oracle's decision hierarchy.

The actual transition from one configuration to the next was chosen at random from:

- the set of actions with the minimum approximate cost with a probability of 0.1;
- from the set of legal actions with a probability of 0.9.

This process was carried out separately for projective and non-projective sentences. The outcomes of this analysis are presented in Tables 6.3 and 6.4.

[1]An average of the number of words between a token and its head.

[2]Considering all configurations encountered while parsing all training sentences, this indicates how many times the approximate minimum cost actions set was included in the exact one

PROJECTIVE	# sentences	% inclusion[2]	Mean Spearman	Std Spearman
en_lines	2921	**100.0:** 99.22, **66.66:** 0.10, **50.0:** 0.60, **33.33:** 0.06, **0.0:** 0.02	0.996	0.043
ko_kaist	17966	**100.0:** 98.78, **66.66:** 0.27, **50.0:** 0.80, **33.33:** 0.10, **0.0:** 0.05	0.991	0.072
grc_proiel	9400	**100.0:** 99.32, **66.66:** 0.16, **50.0:** 0.45, **33.33:** 0.05, **0.0:** 0.02	0.995	0.044
grc_perseus	4272	**100.0:** 99.25, **66.66:** 0.21, **50.0:** 0.49, **33.33:** 0.03, **0.0:** 0.02	0.995	0.045

Table 6.3: Comparison on projective sentences

NON-PROJECTIVE	# sentences	% inclusion	Mean Spearman	Std Spearman
en_lines	253	**100.0:** 98.74, **66.66:** 0.21, **50.0:** 0.92, **33.33:** 0.06, **0.0:** 0.07	0.988	0.071
ko_kaist	5042	**100.0:** 98.00, **66.66:** 0.32, **50.0:** 1.31, **33.33:** 0.21, **0.0:** 0.16	0.961	0.139
grc_proiel	5612	**100.0:** 98.60, **66.66:** 0.23, **50.0:** 0.91, **33.33:** 0.18, **0.0:** 0.08	0.981	0.092
grc_perseus	7202	**100.0:** 98.38, **66.66:** 0.28, **50.0:** 1.00, **33.33:** 0.25, **0.0:** 0.09	0.977	0.102

Table 6.4: Comparison on non-projective sentences

The results indicate that the approximate oracle closely aligns with the exact oracle in determining the set of minimum cost actions. Furthermore, the approximate oracle has the advantage of being linear in time complexity, making it an efficient and effective trade-off for computational purposes.

6.3 Performance

In our dependency parsing model, we employed the architecture developed by Kiperwasser et al. (2016) [8], which utilizes a Bidirectional LSTM (BiLSTM) to derive rich word-level feature representations within sentences. To guide the training process, a Multi-Layer Perceptron (MLP) is employed, which leverages imitation learning techniques, specifically learning to imitate the expert-provided oracle decisions.

The BiLSTM examines sentences in both forward and reverse directions, integrating context from both preceding and subsequent words to inform its understanding of each word's role. This comprehensive context is crucial for the MLP when it scores potential dependency arcs, allowing it to assess the likelihood of a syntactic connection between word pairs.

A significant benefit of this approach is its avoidance of extensive feature engineering and the need for intricate syntactic inputs. Instead, it harnesses the LSTM's inherent ability to encapsulate and utilize long-distance dependencies and sentence structures, simplifying the learning process.

There are many models out there that can provide even higher accuracies, such as BERT, but for this specific situation, we chose the BiLSTM-based model because it trains faster and has good enough performance for this application.

In our training routine, we've adapted Goldberg and Nivre (2012) [5] online training with a dynamic oracle (represented in Algorithm 3 of their paper), incorporating elements from Aufrant et al. (2018) [2] to handle non-projective sentences by considering transitions with minimum cost, as opposed to solely zero-cost transitions. Furthermore, we've deviated from the their update mechanism too: rather than updating the model with each prediction outside the minimum cost set, we accumulate predictions and their corresponding gold la-

with the specified percentage. Examples:
33.33 = only a third of the actions in the approximate set was also present in the exact one;
100.0 = all actions in the approximate set were in the exact one too.

bels throughout the batch, allowing for a singular, comprehensive model update at the batch's conclusion exploiting cross-entropy loss.

Algorithm 3 Modified training loop with a dynamic oracle

```
 1: $w \leftarrow 0$
 2: for $I = 0$ to ITERATIONS do
 3:     for batch in dataset do
 4:         for sentence $x$ with gold tree $G_{\text{gold}}$ in batch do
 5:             $c \leftarrow c_s(x)$
 6:             while $c$ is not terminal do
 7:                 $t_p \leftarrow \arg\max_t w \cdot \phi(c,t)$
 8:                 $MIN\_COST \leftarrow \{t | argmin_t o(t; c; G_{\text{gold}})\}$
 9:                 $t_o \leftarrow \arg\max_{t \in MIN\_COST} w \cdot \phi(c,t)$
10:                 if $t_p \notin MIN\_COST$ then
11:                     Save the couple $(t_p, t_o)$
12:                 end if
13:                 $t_n \leftarrow$ CHOOSE_NEXT$(I, t_p, MIN\_COST)$
14:                 $c \leftarrow t_n(c)$
15:             end while
16:         end for
17:     end for
18:     Update the model with the collected couples $(t_p, t_o)$
19: end for
20: return $w$

    function CHOOSE_NEXT$_{\text{AMB}}$ (I, t, MIN_COST)
21: if $t \in MIN\_COST$ then
22:     return $t$
23: else
24:     return RANDOM_ELEMENT$(MIN\_COST)$
25: end if

    function CHOOSE_NEXT$_{\text{EXP}}$(I, t, MIN_COST)
26: if $I > k$ and RAND() $> p$ then
27:     return $t$
28: else
29:     return CHOOSE_NEXT$_{\text{AMB}}(I, t, MIN\_COST)$
30: end if
```

We assessed the model's performance, measured by the Unlabeled Attachment Score (UAS), across implementations using static, non-deterministic, and dynamic oracles. Due to resource constraints, we could not train the model to imitate the exact dynamic oracle and therefore couldn't directly compare its efficacy with our approximation. Nonetheless, we anticipate that a model trained with the exact dynamic oracle would outperform our approximate oracle-based model.

	Static oracle (only projective, no approximation)	Non Deterministic oracle	Dynamic oracle
en_lines	**74.6**	73.4	73.8
ko_kaist	**70.0**	66.3	65.3
grc_proiel	**64.5**	63.7	64.3
grc_perseus	41.7	44.4	**47.1**

Table 6.5: Comparison of UAS scores across different oracles

The analysis of the UAS with respect to different kinds of sentences in the dataset offers interesting observations regarding the efficacy of dynamic oracles in parsing. Results show that datasets with a higher ratio of non-projective sentences benefit more from the use of dynamic oracles. This is because projective and non-projective sentences are equally informative, but while a dynamic oracle can make use of non-projective sentences, effectively increasing the size of training data, a static oracle cannot exploit this type of data.

The further discussion goes to how parser performance would improve if the Arc-Standard parser had been an arc decomposable: in that case, a linear exact dynamic oracle could be built and it would probably do better. This hypothesis is based on the work of Aufrant et al. (2018) [2], who studied dynamic oracles for arc-eager parsers. They traced the progression from static to non-deterministic, and ultimately to dynamic oracles, observing performance improvements at each stage.

However, the reality for the Arc-Standard parser deviates from this ideal scenario: the non-deterministic and dynamic oracles for this kind of parser rely on an approximate cost function. While this approximation closely aligns with what an exact non-linear dynamic oracle would predict, it remains an approximation. Consequently, significant performance gains are only noticeable when the dataset has a very high percentage of non-projective sentences. The threshold for observing these gains, based on the examined datasets, is at least 60%.

7

An alternative approach: Reinforcement Learning

This chapter discusses the application of Reinforcement Learning (RL) to dependency parsing in NLP. RL, particularly suited for complex decision-making tasks like parsing, learns through trial-and-error with feedback in the form of rewards or penalties. The chapter delves into Markov Decision Process as the foundation of RL, detailing its components and its relevance to parsing configurations and transitions. It highlights the exploration-exploitation trade-off in RL, crucial for handling non-projective dependency structures in parsing. General Policy Iteration is introduced, encompassing Dynamic Programming, Monte Carlo, and Temporal Difference learning approaches, each varying in policy evaluation and improvement methods. Deep Reinforcement Learning, which integrates deep neural networks with RL, is explored for its capability to handle high-dimensional spaces in parsing. The chapter concludes by emphasizing the potential of the Deep Q-Network model, particularly with an Arc-Standard parser, as an effective tool in dependency parsing, offering an adaptable, self-learning approach to improve parsing performance, especially in handling non-projective sentences.

7.1 Why apply Reinforcement Learning to Dependency Parsing?

RL is a machine learning paradigm that deals with problems where an agent interacts with an environment to achieve a specific goal. Instead of relying on labeled training data, RL involves learning through trial-and-error, with the agent receiving feedback from the environment in the form of rewards or penalties.

When applied to dependency parsing, which is the task of determining syntactic relationships between words in a sentence, RL offers several advantages. Dependency parsing often involves a complex decision space, and RL excels in such scenarios where traditional algorithms may struggle. RL models are dynamic and adaptable, making them well-suited for handling diverse sentence structures and linguistic phenomena commonly encountered in natural language processing tasks. One specific challenge in dependency parsing is dealing with non-projective dependency structures, where word relationships do not strictly follow a left-to-right or right-to-left order. RL, particularly when combined with exploration-exploitation trade-offs, can effectively address this challenge. The exploration-exploitation trade-off described by Goldberg and Nivre (2012) [5] and (2013) [6] is inherent in RL approaches, allowing the parser to dynamically adapt its strategy, explore alternative paths, and learn from mistakes: this dynamic behavior enhances adaptability and contributes to the parser's ability to handle non-projective dependencies. However, it appears that Goldberg and Nivre, perhaps due to a lack of familiarity with the RL framework, implicitly introduced several of its central ideas without explicit acknowledgment of their roots. This observation leads to the practical motivation for adopting a more rigorous and methodical application of RL strategies.

Additionally, RL enables end-to-end learning, allowing models to directly learn from input data without the need for explicit feature engineering. This approach also facilitates the integration of global information across the entire sentence, enhancing the model's ability to capture complex dependencies.

Moreover, RL models exhibit flexibility and generalization, making them suitable for various languages and parsing tasks. The capacity of RL to adapt to different linguistic structures contributes to its effectiveness in NLP applications.

Before delving into a specific application of RL in dependency parsing, it's essential to understand some basic concepts of this approach. These may include terms like rewards, policies, exploration-exploitation trade-offs, and the overall framework of RL algorithms. This foundational knowledge provides a basis for comprehending how RL is employed in the context of dependency parsing and the specific strategies utilized to address the challenges inherent in this linguistic task. This chapter is based on the second edition of Sutton and Barto's book: *Reinforcement Learning: An Introduction* [10].

7.2 Markov Decision Processes

A Markov Decision Process (MDP) is a mathematical framework used in the field of RL to model decision-making in situations where an agent interacts with an environment. Specifically, it is a tuple $\langle S, A, P, R, \gamma \rangle$ used to represent the following system:

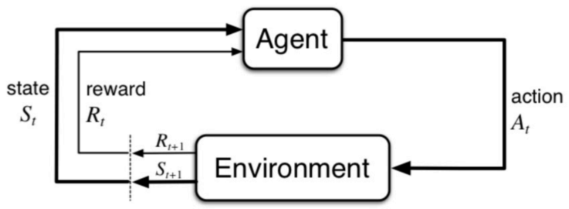


Figure 7.1: Reinforcement Learning dynamics

- S is the set of states that the agent can occupy;
- A is the set of actions that the agent can execute;
- P is the state transition probability matrix: for each state-action pair ($S_t = s, A_t = a$), it indicates the probability that the agent moves to the new state $S_{t+1} = s'$;
- $R : S \times A \to \mathbb{R}$ is the reward function: given the pair ($S_t = s, A_t = a$), it returns a reward for performing action a in state s;
- $\gamma \in [0, 1]$ is a discount factor that indicates how much importance we want to give to future rewards.

The MDP serves as the starting point for RL: the first thing to do is to identify these five elements. In the case of dependency parsing, we have:

- S is the set of configurations in which the parser can find itself;
- A is the set of transition operations that the parser can perform;

- P is fully determined by a function: given a configuration-transition operation pair, the new configuration is uniquely determined, so the transition is not stochastic;
- R can be calculated based on the number of edges no longer reachable due to the executed action or it can be a function of the number of arcs correctly generated.

The key assumption in an MDP is the Markov property, which states that the future state depends only on the current state and action, not on the sequence of events that preceded them. This property allows the modeling and computation of the system's behavior.

The agent's goal in an MDP is typically to find an optimal policy π^* that maximizes the expected cumulative reward over time. A policy is a strategy or a mapping that defines the agent's decision-making behavior: it specifies which action to take in each state.

7.2.1 Environment and Agent

The constitutive elements of the environment are P and R. In particular,

$$R = \mathbb{E}_\pi[R_{t+1}|S_t = s, A_t = a]$$

meaning that the reward function can be interpreted as the expected value of the reward provided by the environment following policy π from the state-action pair $(S_t = s, A_t = a)$.

The constitutive elements of an agent are S, A, policy $\pi(a|s)$, state value function $v_\pi(s)$, and state-action value function $q_\pi(s,a)$:

- the policy is a probability distribution over actions given the current state s:
$$\pi(a|s) = \Pr(A_t = a|S_t = s)$$
- the value of a state is evaluated through the state value function:
$$v_\pi(s) = \sum_a \pi(a|s)[R(a|s) + \gamma \sum_{s'} P(s'|s,a)v_\pi(s')]$$
- the value of a state-action pair is computed by the state-action value function:
$$q_\pi(s,a) = R(a|s) + \gamma \sum_{s'} P(s'|s,a) \sum_{a'} \pi(a'|s')q_\pi(s',a')$$

Essentially, the policy π guides the agent suggesting which action to take based on evaluations made through the value function $v_\pi(s)$ or $q_\pi(s,a)$.

The agent's goal is to maximize the sum of rewards over an episode, which is a sequence of states from an initial state to a final state. π, $v_\pi(s)$, and $q_\pi(s, a)$ are tools used by the agent to achieve this goal.

7.3 Generalized Policy Iteration

To solve a RL problem, the Generalized Policy Iteration (GPI) process is usually executed, which involves alternating between prediction and control phases:

1. Prediction (Policy Evaluation): assign values to visited states (or executed state-action pairs) under a certain policy π. For example, a greedy policy selects in each state the action that leads to the next state with the highest $v_\pi(s')$ value or selects the action with the highest $q_\pi(s, a)$ value.

2. Control (Policy Improvement): after evaluating the states (or state-action pairs) visited, update the policy π. In practice, update the probability distribution based on the $v_\pi(s')$ or $q_\pi(s, a)$ values calculated during the prediction phase.

The alternation between prediction and control is stopped when the values of the value function start to stabilize (i.e. the variation from one iteration to another of prediction is below a fixed threshold).

During the prediction phase, the exploration-exploitation trade-off is implemented. This consists in allowing the agent to choose between exploiting the knowledge of the environment it already possesses by following actions recommended by the policy or exploring new states through state-action combinations not yet tried. It is noteworthy that in RL, the agent requires both exploration and exploitation: without exploration, it might miss the chance to find advantageous state-action combinations, and without exploitation, it would risk continually seeking new combinations without capitalizing on the experience gained, of which the policy π is the bearer.

Generally speaking, the epsilon-greedy strategy serves as a policy to effectively navigate the exploration-exploitation trade-off in RL. At the outset, when the exploration is crucial for building an understanding of the environment, the strategy sets the exploration probability, denoted by ε, to a high value, typically 1.0.

This implies that:

- With a probability of $1 - \varepsilon$: Exploitation occurs, where the agent selects the action associated with the highest state-action pair value, maximizing its current knowledge.

- With a probability of ε: Exploration takes place, involving the selection of a random action to introduce variability and discover potential new insights.

During the initial stages of training, the high value of ε results in a substantial likelihood of exploration. This is beneficial for the agent to explore the environment and gather information. However, as the training progresses and the estimates of the value functions improve in accuracy, the epsilon value is systematically reduced. This reduction reflects the diminishing need for extensive exploration and the increasing emphasis on exploitation.

As a consequence, over time, the probability of exploration decreases, allowing the agent to rely more on its accumulated knowledge for decision-making. The epsilon-greedy strategy thus dynamically adapts, striking a balance between exploration and exploitation throughout the learning process.

7.3.1 EXAMPLE: Gridworld

In Figure 7.2 is represented a classic RL gridworld example: our agent starts from any square in the grid with the objective of reaching one of the dark squares. Each movement incurs a reward of -1 and when the agent reaches a dark square, it receives a positive reward of +10.

On the left column, estimates of the state values $v(s)$ are reported during the policy evaluation (prediction) phase, while on the right, the policy π corresponding to the update performed in the policy improvement (control) phase based on the values of $v(s)$ in the corresponding row is shown.

At iteration $k = 0$, the agent knows nothing about the environment, so all state values are null, and the policy is random. At $k = 1$, the agent starts to understand that states near the dark squares are ”advantageous”, and certain actions in these states are better than others. After updating the $v(s)$ values of these advantageous states, the policy will be updated accordingly, guiding the agent to move accordingly. This process continues iteratively.

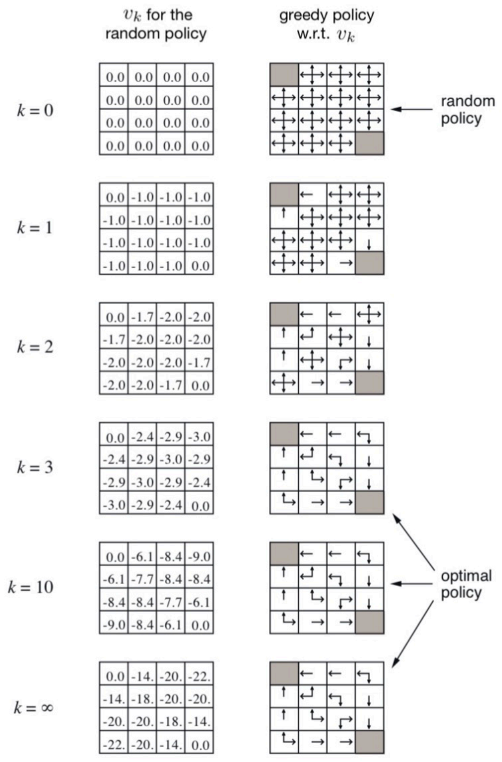


Figure 7.2: Gridworld

In each iteration, the agent refines its understanding of the environment, updating state values and the policy accordingly. The policy guides the agent to make more informed decisions as it learns from its interactions with the environment.

7.3.2 Approaches to GPI

There are three fundamental approaches to GPI, here they are briefly and simplistically presented:

- **Dynamic Programming (DP):** this is illustrated in the Gridworld example. In each prediction iteration, all states are evaluated by averaging the values of states reached by taking an action from the initial state. In each control iteration, the policy is updated for each state. However, problems addressed by RL often have state (or state-action) spaces that are too large for the type of computations required by DP. Subsequent approaches have the same goal as DP but achieve it with considerably lower computational cost.
- **Monte Carlo (MC):** instead of updating each state by considering all possible actions and evaluating all possible reached states, MC performs a set of episodes from start to finish, evaluating only the encountered states

(or executed state-action pairs) at the end of each episode. With the values calculated in the prediction, the policy can be updated during the control phase, but only for the states that have been visited. It's important to note that MC requires saving the state-action-reward sequences encountered during the episode. To evaluate a state (or state-action pair), we need to know the future rewards that state has led toin other words, we need to backtrack through the episode from the end to the beginning.

- **Temporal Difference (TD):** in contrast of running an entire episode, TD allows the agent to learn step-by-step or even every n steps within an episode without reaching the final state. The main advantage of TD is the ability to learn online, without having to save the state-action-reward sequence as in MC. This is possible by approximating future rewards based on the change in the value of a state (or state-action pair) between iterations of the prediction. This approach has two variants:
  - **n-step TD:** fix a number of steps n to simulate in the episode and evaluate the states (or executed state-action pairs) after taking n steps.
  - **TD(λ) or eligibility traces:** instead of using a single "sub-episode" of n steps, multiple "sub-episodes" with different n-steps can be employed, and the values of encountered states in these episodes are averaged.

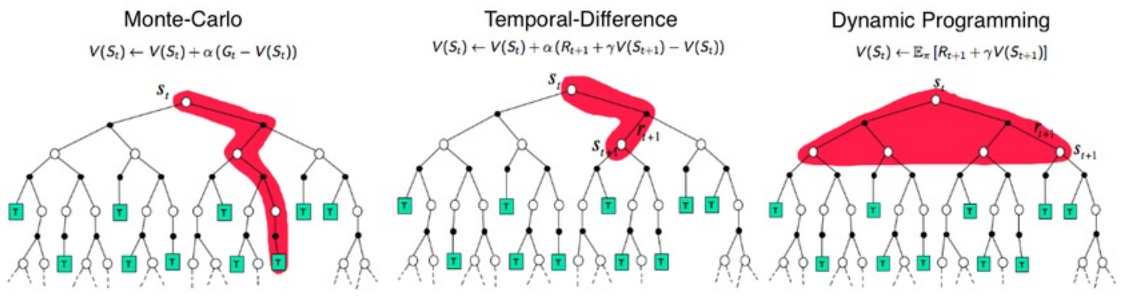

Monte-Carlo
$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$
Temporal-Difference
$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$
Dynamic Programming
$V(S_t) \leftarrow \mathbb{E}_\pi[R_{t+1} + \gamma V(S_{t+1})]$
S_t
r_{t+1}
S_{t+1}
T

Figure 7.3: GPI approaches

7.4 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) stands at the intersection of RL and deep neural networks, harnessing the power of advanced machine learning techniques to address complex problems in high-dimensional and continuous state and action spaces. The integration of deep learning allows DRL models to autonomously discover intricate features and representations from raw input, enabling them to learn and generalize complex behaviors.

Here there is a brief and simplified summary of the main approaches to DRL:

- **Value Approximation:** in value approximation methods, the primary focus is on estimating the value function, representing the expected cumulative reward of taking a specific action in a given state. The introduction of deep neural networks in methods like DQN extends these techniques to handle high-dimensional state spaces. The advantages of value approximation methods include stability and sample efficiency. Q-learning, a prominent example, tends to exhibit lower variance and more stable updates compared to other DRL methods. However, value approximation methods have their disadvantages. They may struggle with exploration, relying on epsilon-greedy strategies that might not be sufficient in complex environments. Additionally, handling continuous action spaces can be challenging without incorporating additional techniques.

- **Policy Gradient:** policy gradient methods, exemplified by algorithms like REINFORCE, take a different approach by directly optimizing the policythe strategy that dictates the agent's actions. These methods parameterize the policy as a neural network, outputting a probability distribution over actions. Policy gradient methods excel in handling continuous action spaces without the need for discretization. They naturally encourage exploration by assigning non-zero probabilities to all actions, making them particularly suitable for scenarios requiring stochastic policies. Nevertheless, policy gradient methods are not without their challenges: they often exhibit high variance in gradient estimates, leading to potentially less stable training. Additionally, achieving sample efficiency may require more training samples compared to some value-based methods.

- **Actor-Critic:** actor-critic methods represent a hybrid approach, combining elements of both value approximation and policy gradient methods. In this framework, two neural networks, the actor (approximating the policy) and critic (approximating the value function), work collaboratively. The actor suggests actions based on the policy, while the critic evaluates these actions using the value function. This combination aims to strike a balance, harnessing the stability of value-based methods and the expressiveness of policy-based methods. The advantages of actor-critic methods include improved stability and sample efficiency compared to standalone policy gradient methods. They can effectively handle continuous action spaces; however, this approach introduces additional complexity, requiring the maintenance of both an actor and a critic network. Fine-tuning parameters becomes crucial to achieve a balance between the contributions of the actor and critic components.

7.5 From Q-learning to the DQN model

Q-learning is a fundamental RL algorithm designed to find an optimal policy for an agent interacting with an environment. The algorithm iteratively updates Q-values based on the equation:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \cdot \left[R_{t+1} + \gamma \cdot \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right].$$

Here, $Q(S_t, A_t)$ represents the Q-value for taking action A_t in state S_t, R_{t+1} is the immediate reward, $\gamma \cdot \max_a Q(S_{t+1}, a)$ is the discounted estimate optimal Q-value of next state, and α is the learning rate. Q-learning aims to learn the optimal Q-values, which, when followed, lead to the agent making decisions that maximize the expected cumulative reward.

Algorithm 4 Q-Learning algorithm

```
1: Initialize $Q(s, a)$ arbitrarily for all state-action pairs except $Q(\text{terminal}, .) = 0$
2: for each episode do
3:     Observe state $S_0$
4:     $t \leftarrow 0$
5:     while $S_t$ not terminal do
6:         Choose action $A_t$ from $S_t$ using a policy derived from $Q$ (e.g., $\epsilon$-greedy)
7:         Take the chosen action $A_t$, observe the next state $S_{t+1}$ and the immediate
           reward $R_{t+1}$
8:         $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \cdot [R_{t+1} + \gamma \cdot \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$
9:         $S_t \leftarrow S_{t+1}$
10:    end while
11: end for
```

The transition from Q-learning to DQN introduces the use of deep neural networks to approximate the Q-function. The Q-function is parameterized by a neural network with weights θ, denoted as $Q(s, a; \theta)$. The update rule for DQN involves minimizing the temporal difference loss:

$$\mathcal{L}_{(\theta)} = \mathbb{E}\left[\left(R_{t+1} + \gamma \cdot \max_{a \in \mathcal{A}_{(S_{t+1})}} Q(S_{t+1}, a; \theta^-) - Q(S_{t+1}, A_{t+1}; \theta)\right)^2\right]$$

where θ^- represents the parameters of a target Q-network with delayed updates. This approximation allows DQN to handle high-dimensional state spaces effectively, enabling its application to complex tasks, where traditional Q-learning might be impractical.

Algorithm 5 DQN training algorithm

```
 1: Initialize replay memory $D$ to capacity $N$
 2: Initialize action-value function $Q$ with random weights $\theta$
 3: Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
 4: for each episode do
 5:     Observe state $S_0$
 6:     $t \leftarrow 0$
 7:     while $S_t$ not terminal do
 8:         With probability $\varepsilon$ select a random action $A_t$
 9:         otherwise select $A_t = \mathrm{argmax}_a Q(S_t, a; \theta)$
10:         Execute action $A_t$ and observe reward $R_t$, and next state $S_{t+1}$
11:         Store transition $(S_t, A_t, R_t, S_{t+1})$ in $D$
12:         Sample random minibatch of transitions $(S_j, A_j, R_j, S_{j+1})$ from $D$
13:         Set
                $$y_j = \begin{cases} R_j \text{ if episode terminates at step } j+1 \\ R_{t+1} + \gamma \cdot \max_{a \in \mathcal{A}(S_{t+1})} Q(S_{t+1}, a; \theta^-) \text{ otherwise} \end{cases} \quad (7.1)$$
14:         Perform a gradient descent step on $\left(y_j - Q(S_{t+1}, A_{t+1}; \theta)\right)^2$ with respect
            to network parameters $\theta$
15:         Every $C$ steps reset $\hat{Q} = Q$
16:     end while
17: end for
```

The DQN training algorithm comprises two fundamental phases, each playing a pivotal role in the learning process. In the initial phase, known as the Sampling Phase (lines 8-12), the agent actively interacts with the environment by executing actions, and the resulting experiences are systematically stored in a designated replay memory. These experiences are encapsulated as tuples, including the state, action taken, received reward, and the subsequent state. This process not only facilitates exploration of the environment but also establishes a diverse and representative dataset within the replay memory, crucial for effective learning.

Subsequently, the algorithm enters the Training Phase (lines 13-16), wherein the agent randomly selects a small batch of experience tuples from the replay memory. This batch is then utilized to update the Q-values through a gradient descent update step. By learning from randomly sampled batches, the model breaks the temporal correlation between consecutive experiences, fostering stability and efficiency in the learning process. The objective is to minimize the temporal difference error between predicted and target Q-values, refining the Q-network's parameters over iterations.

This iterative interplay between the Sampling and Training Phases enables the DQN algorithm to adapt and improve its Q-value estimates over time. By leveraging the stored experiences in the replay memory, the model learns to make informed decisions, ultimately leading to the acquisition of an effective policy that maximizes cumulative rewards in complex and dynamic environments.

In the context of DQN, training stability can be a significant challenge, primarily due to the amalgamation of a non-linear Q-value function represented by a neural network and the incorporation of bootstrapping, where targets are updated using existing estimates rather than complete returns. This combination may lead to training instability, therefore, to counteract this issue and enhance the stability of training, three key solutions are commonly implemented:

- **Experience Replay:** Experience Replay is employed with the aim of making more efficient use of past experiences. Rather than training the neural network on consecutive experiences, this technique involves storing a history of experiences in a replay buffer and randomly sampling a batch of these experiences during each training iteration. This breaks the temporal correlation between consecutive experiences, providing a more diverse set of training samples and promoting stability.

- **Fixed Q-Target:** the utilization of a Fixed Q-Target serves the purpose of stabilizing the training process. To achieve this, a separate target network with fixed parameters is introduced. This target network is updated less frequently compared to the main Q-value estimation network. The less frequent updates reduce the correlation between the target and predicted Q-values during training, ultimately stabilizing the learning process and preventing rapid and oscillatory updates.

- **Double Deep Q-Learning:** Double Deep Q-Learning addresses the problem of Q-value overestimation, which is a common issue in DQN. By employing two separate neural networks, one for selecting actions (policy network) and another for evaluating Q-values (target network), this approach aims to mitigate the overestimation bias. The decoupling of action selection from Q-value evaluation helps in providing more accurate Q-value estimates, leading to improved learning and decision-making in the RL setting.

These three strategies, when integrated, contribute synergistically to stabilizing the training of Deep Q-Learning models. This not only mitigates the inherent challenges associated with the combination of non-linear Q-value functions and bootstrapping but also enhances the robustness of DQN models in effectively learning optimal policies in complex environments.

7.6 The DQN model for dependency parsing

Returning to the topic of dependency parsing, our chosen model is the DQN. Specifically tailored for transition-based dependency parsing with an Arc-Standard parser, DQN excels in managing sequential decision-making tasks. In the context of dependency parsing, the primary objective is to construct a syntactic tree by executing a sequence of parser transitions: DQN's state-action framework proves to be a suitable representation for the evolving parser configuration.

The strength of DQN lies in its ability to approximate the Q-function, aligning perfectly with the task of evaluating the expected cumulative reward for each transition within a given state. Moreover, the utilization of deep neural networks by DQN is advantageous for effectively handling the inherent high-dimensional state spaces encountered in dependency parsing. As shown by Yu et al. (2018) [12], the model's generalization across linguistic contexts, complemented by features like experience replay, Dueling Deep Q-Network (DDQN) and averaged DQN, enhances its effectiveness in learning diverse transition patterns.

Furthermore, DQN presents an intriguing alternative to dynamic oracles when dealing with non-projective sentences in dependency parsing. By learning directly from environmental interactions, DQN can adapt and discover effective transition strategies for non-projective sentences without relying on explicitly constructed oracles. This self-learning capability proves particularly beneficial in scenarios where dynamic oracles may be impractical or challenging to accurately define (as stated in Chapter 4, this is the case for the Arc-Standard parser). The RL paradigm of DQN facilitates natural exploration and exploitation of the solution space, potentially leading to improved performance in handling non-projective syntactic structures during the parsing process.

7.6.1 Averaged DQN

The averaged DQN technique (Anschel et al., 2016 [1]), represents a significant stride in the realm of RL, particularly addressing the challenges of volatility and instability that are commonly observed in traditional DQNs. This approach is rooted in the concept of integrating multiple Q-value estimators over time, ensuring a smoother and more reliable policy evaluation and decision-making process.

In standard DQN settings, the Q-value estimates are typically updated at each step or after a fixed number of steps, and these immediate estimates are directly utilized for guiding actions and further updates. However, this approach can lead to considerable oscillations in the learning trajectory: the reason lies in the nature of Q-value updates themselves, which can be substantial at times, causing sharp shifts in policy direction. This is where the averaged DQN technique comes into play, offering a more measured and stable alternative. By maintaining a record of not just the current Q-value estimator but also an array of past estimators, the averaged DQN method ensures that the policy is informed by a more balanced perspective, one that is shaped by an aggregation of past learning experiences as well as the present.

The real prowess of the averaged DQN technique lies in its averaging mechanism. This mechanism doesn't just blunt the edges of variance in Q-value estimates; it fundamentally reshapes the learning landscape. Each Q-value estimator in the series is updated akin to a standard DQN. However, when it comes to policy determination or making decisions, the averaged DQN doesn't rely on the latest snapshot of learning; instead, it consults the averaged wisdom of multiple past estimators. This approach significantly dampens the volatility typically associated with Q-value updates, leading to a more consistent and reliable decision-making process.

Moreover, the technique addresses the notorious issue of catastrophic forgetting, a scenario where a network suddenly loses the information it previously learned upon acquiring new knowledge. The averaged DQN, with its diverse repository of estimators, ensures that the influence of any single update is appropriately moderated. This preservation of knowledge across a spectrum of learning stages fortifies the agent against abrupt losses of valuable information, ensuring a more comprehensive and resilient learning journey.

The stabilization of the learning curve is another hallmark of the averaged

DQN. By tempering the sharp swings and fluctuations in Q-value estimates through its averaging process, the technique fosters a smoother and more predictable progression in learning. This stability is not just beneficial for the consistency of policy improvement; it also instills a sense of reliability in the learning outcomes, a quality that's immensely valuable in environments where decisions carry significant consequences.

Furthermore, averaged DQN inherently counters the tendency of Q-value overestimation, a prevalent challenge in standard DQN setups. The averaging process naturally offsets the extremes, ensuring that both overestimated and underestimated Q-values are harmonized towards a more balanced estimation. This balance is crucial, not just for the accuracy of value estimation but also for maintaining a healthy equilibrium between exploration and exploitation. With more stable and dependable Q-value estimates, the agent is less prone to being swayed by erratic estimations of action values, promoting a more measured and strategic approach to exploring the environment and capitalizing on the learned knowledge.

In essence, the averaged DQN technique enriches the landscape of RL by introducing a method that emphasizes stability, reliability, and a comprehensive integration of learning experiences. By leveraging the collective insight of multiple Q-value estimators, it paves the way for smoother learning trajectories, fortified retention of knowledge, and more judicious decision-making, especially in complex and dynamic environments where the clarity and dependability of every learning step are paramount.

7.6.2 Dueling DQN

The DDQN technique (Wang et al., 2016 [11]) brings a nuanced approach to learning in reinforcement learning scenarios, particularly distinguishing itself from the traditional DQN through its unique architectural design. This architecture meticulously separates the estimation of two critical components: the value of being in a given state, and the additional value derived from taking a particular action in that state.

In a traditional DQN setup, the action value function (Q-value) for each possible action in a given state is straightforwardly estimated. This approach is denoted as $Q(s, a)$ and doesn't explicitly differentiate between the intrinsic value of the state s itself and the value of taking action a in that state. The DDQN addresses this by splitting the network into two distinct streams beyond a shared set of initial layers. One stream, known as the "Value Stream", is dedicated to assessing the value of being in a particular state s, yielding $V(s)$. The other stream, termed the "Advantage Stream", computes the advantage of each action a in state s, represented as $A(s, a)$. This advantage quantifies how much better it is to take a particular action compared to others on average.

The central idea behind the DDQN is to allow the estimation of the state's value to be decoupled from the advantages of individual actions, thereby enabling a more focused and potentially more accurate estimation of both. However, merging these two streams to get the final Q-value isn't straightforward. A naive addition of $V(s)$ and $A(s, a)$ could lead to identifiability issues, where the same Q-value could be decomposed into multiple combinations of values and advantages. To circumvent this, the combined Q-value is calculated by adding the state value, $V(s)$, to the advantage of action a, $A(s, a)$, but then subtracting the average advantage of all actions in state s, mathematically expressed as:

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|A|} \cdot \sum_{a'} A(s,a') \right)$$

In this equation, $\frac{1}{|A|} \cdot \sum_{a'} A(s, a')$ is the average advantage of all possible actions in state s, ensuring that the advantages have a zero mean for each state. This adjustment not only addresses the identifiability problem, but also stabilizes the learning by keeping the estimation of state values consistent across different actions.

The decomposition of Q-values into state values and action advantages brings several theoretical and practical benefits. It helps the network to focus on what really matters, distinguishing between the value of different states, and understanding the impact of each action within those states. The architecture is particularly advantageous in scenarios where the choice of action does not significantly affect the state's outcome, allowing the model to spend more computational resources on understanding the state's value. On the other hand, in situations where the choice of action is crucial, the network can effectively distinguish the relative worth of each action, leading to more informed and strategic decision-making.

In conclusion, the DDQN, with its dual-stream architecture, brings a sophisticated perspective to the estimation of Q-values in reinforcement learning.

By separately and effectively estimating the value of states and the advantages of actions, the DDQN enables a more nuanced, stable, and efficient learning process, proving its worth in complex decision-making scenarios.

7.6.3 Proportional Prioritized Experience Replay

The Proportional Prioritized Replay Buffer (PPRB), introduced by Schaul et al. (2015) [9], is a refined strategy in RL that optimizes the training of agents by modifying the traditional methods of storing and retrieving experiences. This technique emphasizes the importance of certain experiences over others during the learning process, based on the premise that not all experiences contribute equally to the agent's learning progression.

Traditionally, experiences are stored in a replay buffer and are sampled uniformly at random when training the agent. However, this approach does not differentiate between the experiences based on their value for learning. Some experiences may occur rarely but provide significant insight into the environment, while others may be frequent yet offer limited new information. The PPRB addresses this imbalance by assigning a priority to each experience, which is reflective of its potential to improve the agent's policy.

In this method, each experience is assigned a priority score based on the magnitude of its TD error, denoted as δ. The TD error is a measure of the difference between the current estimated value of a state-action pair (Q-value) and the improved estimate following an observed transition, calculated as

$$\delta = |r + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a)|.$$

Here, r represents the reward received after taking action a in state s, leading to the next state s', and γ is the discount factor that moderates the importance of future rewards.

The probability of an experience being sampled is made proportional to its priority. Specifically, if p_i represents the priority of the i^{th} experience, then the sampling probability $P(i)$ is given by $P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$. The exponent α is a hyperparameter that governs the level of prioritization, with $\alpha = 0$ yielding uniform random sampling and larger values increasing the focus on high-priority experiences.

However, prioritizing experiences in this way introduces bias because it changes the distribution from which experiences are drawn. To correct for this, Importance-Sampling (IS) weights are applied during the update of the Q-values. The IS weight for the i^{th} experience is computed as $w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^{\beta}$, where N is the size of the replay buffer and β is a hyperparameter that adjusts the degree of bias compensation. With $\beta = 0$, there is no compensation (resulting in high bias), and with $\beta = 1$, the compensation is full (leading to high variance).

To ensure that the sampling probabilities remain aligned with the potential for learning, the priorities of experiences are updated after each learning step based on the new TD errors. This dynamic adjustment of priorities ensures that the sampling strategy continues to focus on the most informative experiences as the agent's policy evolves.

The PPRB offers numerous advantages. It fosters a more focused learning approach, directing the agent's attention to experiences that are expected to provide the most valuable learning insights. This leads to improved sample efficiency, as the agent can achieve better performance with fewer interactions by revisiting critical experiences more frequently. The use of IS weights also helps stabilize the learning process by preventing large updates from dominating the agent's learning trajectory. Furthermore, this technique ensures that rare but pivotal experiences are not lost in the sea of more common occurrences, providing a balanced learning environment that appreciates the significance of every encounter in the complex landscape of the agent's experience.

8 Results: part II

To develop our dynamic oracle with the DQN approach, we closely adhered to the methodology outlined by Yu et al. (2018) [12] in their work on reinforcement learning-based dynamic oracles. Our commitment in replicating their framework was unwavering, however due to the absence of their original source code, we autonomously took certain implementation decisions. This situation led to the need for assumptions, particularly in tuning hyperparameters, which were adjusted based on limited experimentation rather than extensive, systematic trials. These provisional choices highlight areas for in-depth analysis in future research endeavors.

In this context, we provide an overview of both the DQN model and the parser training process. Our discussion includes the strategies employed for training the DQN model, focusing on the adjustments made in the absence of specific details from the original study. Additionally, we elucidate the parser training methodology, emphasizing how the DQN model integrates with and enhances this process.

Concluding our discussion, we present an updated comparison (refer to Table 8.1) of the parser's performance with different types of oracle. This comparison critically evaluates Arc-Standard parsers trained through imitation learning, contrasting those that utilize "classic" oracles against those leveraging the DQN model. Our findings aim to shed a light on the effectiveness of the DQN-based approach in improving parsing accuracy and efficiency, providing a comprehensive understanding of its impact compared to traditional methods.

8.1 Training of DQN model and parser

The training procedure of the DQN model is detailed with a focus on handling erroneous states in dependency parsing. The training involves random sampling of training instances that capture erroneous states, which will be crucial in ensuring the model learns proper recovery strategies. This is done through a process which, besides following an ϵ-greedy policy, consists in forking paths with a certain probability by taking a random valid action. This way the model can simulate parser errors, treating each forked path as a new episode starting from the state after the forking action. To enhance sample efficiency, only the first N states in each episode are considered.

This approach ensures that the model learns from all kinds of scenarios, acquiring the knowledge of how to properly lead the parser through the potential errors in parsing. The ϵ-greedy policy, together with forking, simulates a realistic parsing environment where deviations from the optimal path occur, thus training the oracle to anticipate and correct for these deviations. This not only strengthens the parsing system that results in the presence of errors, but generally enhances the ability of recovery from mistakes and hence improves parsing accuracy and efficiency.

The input features to the DQN are simple yet effective: binary indicators represent the position of the gold head for the first ten tokens in both the stack and the buffer. Additionally, binary features indicate whether the gold head of a token is missed and whether all pending gold dependents of a token have been collected. This is done to make sure that the DQN model is presented with a distilled essence of the parsing configuration and can thus focus on learning the optimum parsing actions without being overwhelmed by the complexity of lexical data.

Furthermore, the inclusion of the five previous actions leading to the current state and all valid actions in this state as part of the input features introduces a dynamic element to the model, allowing it to consider both historical and potential future actions in its decision-making process. This design choice not only enhances the model's ability to anticipate and recover from parsing errors, but also enhances it with a sophisticated understanding of the parsing trajectory, enabling it to navigate the complex landscape of dependency parsing with greater finesse and strategic foresight.

The training process for the parser is relatively straightforward. After training the oracle (DQN) to manage incorrect states or configurations, the parser's training follows an imitation learning approach, leveraging the expertise of the DQN model: at each configuration both the parser and the oracle are required to predict the next action. The parser produces a distribution probability across its transitions, while the oracle's prediction indicates the actual best move: through the cross entropy loss computation and its backward propagation the neural network modeling the parser is updated.

8.2 Performance

The performance outcomes achieved from our study align with the findings presented by Yu et al. (2018) [12]. This comparison reveals that, while the reward function employed by Yu and colleagues boasts a more universal application (capable of training a single general oracle to support various parser types, including Arc-Standard, Swap, Attardi, and Arc-Hybrid), our reward function is tailor-made for the Arc-Standard parser and, theoretically, should deliver superior performance for such parser. However, the expected enhancement in performance was not fully realized in our results.

One plausible reason for this discrepancy, as we highlighted at the start of this chapter, is the lack of a comprehensive hyperparameter tuning study in our approach. The absence of meticulous optimization of the model's hyperparameters might have hindered achieving the potential peak performance of our parser. Identifying and addressing this gap through a focused hyperparameter optimization effort could significantly boost the model's efficiency and effectiveness.

Despite these challenges, the RL strategy we employed marks a promising direction, especially for parsers that lack the arc decomposition property. As previously mentioned, this feature is crucial for developing linear dynamic oracles capable of handling non-projective sentences. Our RL-based approach stands out as a viable alternative, demonstrating the potential of reinforcement learning in enhancing parser frameworks that traditionally struggle with such linguistic structures.

	Static oracle (only projective, no approximation)	Non Deterministic oracle	Dynamic oracle	DQN (only projective)	DQN (also non-projective)
en_lines	74.6	73.4	73.8	74.2	**75.0**
ko_kaist	70.0	66.3	65.3	71.5	**73.1**
grc_proiel	64.5	63.7	64.3	64.4	**67.5**
grc_perseus	41.7	44.4	47.1	48.2	**49.3**

Table 8.1: Comparison of UAS scores among ”classic” and RL-based oracles

9 Conclusions and Future Works

This thesis has embarked on a comprehensive exploration of transition-based dependency parsing, delving into the intricacies of parser types, the role of oracles, and the innovative application of RL to improve parsing strategies. Through meticulous analysis and experimentation, this work has shed light on the capabilities and limitations of various parsing systems, particularly focusing on the Arc-Standard parser, and has highlighted the potential of RL in enhancing dependency parsing performance.

The journey began with a detailed examination of Arc-Standard, Arc-Eager, and Arc-Hybrid parsers, each presenting unique approaches to constructing dependency trees. The subsequent discussion on static, non-deterministic, and dynamic oracles underscored their crucial role in guiding parsers through optimal transitions. The introduction of RL, specifically through the DQN model, marked a pivotal turn in this exploration, offering a self-learning mechanism capable of navigating the challenges of non-projective dependency structures.

The application of the DQN model to the Arc-Standard parser, as detailed in Chapter 7, has opened new avenues for parsing strategies. By harnessing the power of deep neural networks and leveraging techniques such as averaged DQN, dueling DQN and proportional prioritized experience replay, the DQN model demonstrated its potential in accurately parsing complex linguistic structures. However, the full impact of RL on dependency parsing, particularly in handling non-projective sentences, remains a promising field of exploration, with encouraging initial results.

9.1 Future Directions

9.1.1 Expanding Reinforcement Learning Experiments

Future research could further explore the application of RL to dependency parsing by conducting extensive experiments across a wider array of parser types. This would not only validate the performance gains of the DQN model, but also uncover challenges and opportunities for RL in parsing: if the RL exploration-exploitation paradigm is actually superior to the one proposed in ”traditional” NLP literature for dynamic oracles, considerable improvements in parsing accuracy could be obtained. Additionally, experimenting with other RL algorithms, such as Proximal Policy Optimization (PPO) or Soft Actor-Critic (SAC), could offer insights into the most effective RL approaches for dependency parsing.

9.1.2 Incorporating Advanced Neural Network Architectures

Exploring advanced neural network architectures, such as transformers, within the RL framework for dependency parsing, could significantly enhance the model’s ability to understand and represent complex linguistic patterns. This approach could leverage the self-attention mechanism to better capture long-distance dependencies, almost certainly surpassing the performance of our developed models.

9.1.3 Extensive Experiment with the Exact Dynamic Oracle for the Arc-Standard Parser

A critical avenue for future research that was not explored in this thesis, due to resource constraints, is conducting an extensive experimentation with the exact dynamic oracle for the Arc-Standard parser. This testing would be essential for establishing a robust baseline that can serve as a benchmark in evaluating our models. The exact dynamic oracle, with its ability to provide the optimal set of transitions for constructing dependency trees, represents the theoretical best performance achievable under the Arc-Standard parsing framework. Implementing and testing the exact dynamic oracle would not only validate the effectiveness of the approximate dynamic oracle and other RL-based approaches discussed in this thesis, but also highlight areas where these methods can be further optimized. By understanding the performance gap between the exact oracle and the proposed models, researchers would identify specific challenges and opportunities for enhancing parsing accuracy, especially in dealing with non-projective sentences.

9.2 Concluding Remarks

This thesis represents a first step in the formal application of RL to dependency parsing, offering a new perspective on tackling the challenges inherent in understanding natural language syntax. The exploration of dynamic oracles, the integration of RL, and the preliminary experiments with the DQN model have laid a solid foundation for future research in this exciting intersection of NLP and RL.

As we consider the next steps, the directions we've proposed hold potential for enriching our understanding and effectiveness in dependency parsing, contributing modestly to the broader field of NLP technologies. These avenues, from conducting comprehensive experiments with dynamic oracles to exploring the nuances of RL and the impact of non-projectivity, represent our commitment to a deeper exploration of how machine learning can enhance our engagement with human language. By embracing these opportunities for innovation and research, we hope to contribute to the ongoing dialogue in NLP, understanding that our efforts are but one part of a much larger community endeavor. It's through this collective pursuit of knowledge and the humble recognition of the vast complexities of language that we can move forward, inch by inch, towards more sophisticated and nuanced computational linguistics tools. Our journey is ongoing, and each small step contributes to the gradual unveiling of the potential machine learning holds in bridging the gap between computational processes and the rich tapestry of human communication.

A Code I

```python
!pip install datasets
!pip install conllu

import torch
import torch.nn as nn
from functools import partial
from datasets import load_dataset

!pip install evaluate

import matplotlib.pyplot as plt
import numpy as np
import copy
import time
import random as rd
```

Code A.1: pip and import

```python
# the function returns whether a tree is projective or not. It is
# currently implemented inefficiently by brute checking every pair
# of arcs.
def is_projective(tree):
  for i in range(len(tree)):
    if tree[i] == -1:
      continue
    left = min(i, tree[i])
    right = max(i, tree[i])

    for j in range(0, left):
      if tree[j] > left and tree[j] < right:
        return False
    for j in range(left+1, right):
      if tree[j] < left or tree[j] > right:
        return False
    for j in range(right+1, len(tree)):
      if tree[j] > left and tree[j] < right:
        return False

  return True

# the function creates a dictionary of word/index pairs: our
# embeddings vocabulary threshold is the minimum number of
# appearance for a token to be included in the embedding list
def create_dict(dataset, threshold=3):
  dic = {}  # dictionary of word counts
  for sample in dataset:
    for word in sample['tokens']:
      if word in dic:
        dic[word] += 1
      else:
        dic[word] = 1

  map = {}  # dictionary of word/index pairs: our embedding list
  map["<pad>"] = 0
  map["<ROOT>"] = 1
  map["<unk>"] = 2 #used for words that do not appear in our list

  next_indx = 3
  for word in dic.keys():
    if dic[word] >= threshold:
      map[word] = next_indx
      next_indx += 1

  return map

# split the dataset in training, validation and testing sets
train_dataset = load_dataset('universal_dependencies', 'en_lines',
    split="train")
dev_dataset = load_dataset('universal_dependencies', 'en_lines',
    split="validation")
test_dataset = load_dataset('universal_dependencies', 'en_lines',
    split="test")

# build a dictionary counting the number of occurrences of each type
    of the dataset
# and a list containing the number of word for each phrase
types = {}
phrase_lengths = []
for dataset in train_dataset, dev_dataset, test_dataset:
  for s in dataset:
    phrase_lengths.append(len(s['tokens']))
    for t in s['tokens']:
      types[t] = types.get(t, 0) + 1

# For unlabeled depedency parsing we only need the gold tree
train_dataset = [sample for sample in train_dataset if 'None' not in
    sample["head"]]
dev_dataset = [sample for sample in dev_dataset if 'None' not in
    sample["head"]]
test_dataset = [sample for sample in test_dataset if 'None' not in
    sample["head"]]
```

Code A.2: dataset management

```
class ArcStandard:
  def __init__(self, sentence):
    # Initialize the parser with the input sentence
    self.sentence = sentence
    # Create a buffer with indices for each word in the sentence
    self.buffer = [i for i in range(len(self.sentence))]
    # Initialize an empty stack for the parsing process
    self.stack = []
    # Initialize a list to store arcs with default -1 (no arc)
    self.arcs = [-1 for _ in range(len(self.sentence))]

    # Perform three shift operations to start the parsing process
    self.shift()
    self.shift()
    # If the sentence has more than two words, perform a third shift
    if len(self.sentence) > 2:
      self.shift()

    # Initialize the loss array for the stack (needed for exact
    dynamic oracle)
    self.loss = [0 for i in range(len(self.stack))]
```

```python

  def shift(self):
    # Take the first item from the buffer and move it to the stack
    b1 = self.buffer[0]
    self.buffer = self.buffer[1:]
    self.stack.append(b1)

  def left_arc(self):
    # Remove the second item from the stack and create an arc from it to the top of the stack
    o1 = self.stack.pop()
    o2 = self.stack.pop()
    self.arcs[o2] = o1
    # Put the top item back on the stack
    self.stack.append(o1)
    # If the stack is too small and the buffer isn't empty, perform a shift
    if len(self.stack) < 2 and len(self.buffer) > 0:
      self.shift()

  def right_arc(self):
    # Remove the top item from the stack and create an arc to the second item
    o1 = self.stack.pop()
    o2 = self.stack.pop()
    self.arcs[o1] = o2
    # Put the second item back on the stack
    self.stack.append(o2)
    # If the stack is too small and the buffer isn't empty, perform a shift
    if len(self.stack) < 2 and len(self.buffer) > 0:
      self.shift()

  def is_tree_final(self):
    # Check if the parsing is complete: one item on the stack and buffer is empty
    return len(self.stack) == 1 and len(self.buffer) == 0

  def print_configuration(self):
    # Print the current stack and buffer states along with loss and arcs
    s = [self.sentence[i] for i in self.stack]  # Convert indices to words for printing
    b = [self.sentence[i] for i in self.buffer]  # Convert indices to words for printing
    print(s, b)
    print(self.loss)
    print(self.arcs)
```

Code A.3: Arc-Standard Parser

```python
class StaticOracle:
  def __init__(self, parser, gold_tree):
    # Initialize the oracle with a parser instance and a gold parse tree
    self.parser = parser
    self.gold = gold_tree

  def is_left_arc_gold(self):
    # Check if the left arc action is correct according to the gold tree
    s1 = self.parser.stack[-1]  # Top of the stack
    s2 = self.parser.stack[-2]  # Second item from the top of the stack

    # If s2 is the parent of s1 in the gold tree, left arc is correct
    return self.gold[s2] == s1

  def is_right_arc_gold(self):
    # Check if the right arc action is correct according to the gold tree
    s1 = self.parser.stack[-1]  # Top of the stack
    s2 = self.parser.stack[-2]  # Second item from the top of the stack

    # If s1 is not the parent of s2, right arc is not correct
    if self.gold[s1] != s2:
      return False

    # Check for any dependents of s1 to the right in the buffer
    for i in self.parser.buffer:
      if self.gold[i] == s1:
        return False

    # Right arc is correct if none of the above conditions are met
    return True

  def is_shift_gold(self):
    # Check if the shift action is correct according to the gold tree
    # Shifting is not possible if buffer is empty
    if len(self.parser.buffer) == 0:
      return False

    # Shifting should not be done if a left or right arc is possible
    if self.is_left_arc_gold() or self.is_right_arc_gold():
      return False

    # Shifting is correct if none of the above conditions are met
    return True
```

Code A.4: Static Oracle

```
class ExactDynamicOracle:
  def __init__(self, parser, gold_tree):
    # Initialize the oracle with the parser and the gold parse tree
    self.parser = parser
    self.gold = gold_tree
    # Compute the initial loss of the parser state compared to the
   gold tree
    self.current_cost = computeLoss(self.parser, self.gold)

  def cost_left_arc(self):
    # Calculate cost for left arc; return infinity if the transition
   is illegal
    if len(self.parser.stack) < 2 or (len(self.parser.stack) == 2 and
     len(self.parser.buffer) != 0) or self.parser.stack[-2] == 0:
      return float('inf')

    # Create a deep copy of the parser to simulate the left arc
   without affecting the original parser
    fake_parser = copy.deepcopy(self.parser)
    # Update loss if the left arc does not align with the gold tree
    if self.gold[fake_parser.stack[-2]] != fake_parser.stack[-1]:
      fake_parser.loss[-1] += 1
    # Merge losses and perform the left arc
    fake_parser.loss[-1] += fake_parser.loss.pop(-2)
    fake_parser.left_arc()
    # Compute the new cost and return the difference
    new_cost = computeLoss(fake_parser, self.gold)
    return new_cost - self.current_cost

  def cost_right_arc(self):
    # Similar logic to cost_left_arc but for right arc transitions
    if len(self.parser.stack) < 2 or (self.parser.stack[-2] == 0 and len(self.parser.buffer) > 0):
      return float('inf')

    fake_parser = copy.deepcopy(self.parser)
    if self.gold[fake_parser.stack[-1]] != fake_parser.stack[-2]:
      fake_parser.loss[-2] += 1
    if len(fake_parser.loss) > 2:
      tmp = fake_parser.loss[-2] + fake_parser.loss[-1]
      fake_parser.loss[-2] = tmp
    else:
      fake_parser.loss[0] += fake_parser.loss[1]

    fake_parser.loss.pop()
    fake_parser.right_arc()

    new_cost = computeLoss(fake_parser, self.gold)

    return new_cost - self.current_cost

  def cost_shift(self):
    # Calculate cost for shift; return infinity if the transition is illegal
    if len(self.parser.buffer) == 0:
      return float('inf')

    fake_parser = copy.deepcopy(self.parser)
    fake_parser.shift()
    fake_parser.loss.append(0)
    new_cost = computeLoss(fake_parser, self.gold)

    return new_cost - self.current_cost

  def provideTransitionCosts(self):
    # Provide costs for all types of transitions
    return [self.cost_left_arc(), self.cost_right_arc(), self.cost_shift()]

# Auxiliary functions for the Exact Dynamic Oracle
def computeLoss(parser, gold):
  b = len(parser.stack)
```

```python
  gamma = parser.stack + parser.buffer
  table = [[{} for _ in range(len(gamma))] for _ in range(len(gamma))]

  # Table initialization
  j = 0
  for i in range(len(gold)):
    if parser.arcs[i] != -1:
      continue
    if j < b: # Initialize table entries for diagonal elements coming from the stack
      table[j][j] = {j: [parser.loss[j], i]}
      j += 1
    elif j < len(gamma) and j >= b: # Initialize table entries for diagonal elements coming from the buffer
      table[j][j] = {j: [0, i]}
      j += 1

  # Fill in the remaining table entries
  for j in range(b - 1, len(gamma)):
    for i in range(j, -1, -1):
      fillTableEntry(table, i, j, gold)

  return table[0][len(gamma) - 1][0][0]

def checkGold(head, dependent, gold):
  if gold[dependent] == head:
    return 0
  else:
    return 1

def fillTableEntry(table, i, j, gold):
  for k in range(i, j):  # Loop over possible split points in the interval [i, j-1]
    for key_left in table[i][k]:
      for key_right in table[k+1][j]:
        value_right = table[i][j].get(key_right, (float('inf'), None))
        value_left = table[i][j].get(key_left, (float('inf'), None))

        loss_left_arc = table[i][k][key_left][0] + table[k+1][j][key_right][0] + checkGold(table[k+1][j][key_right][1], table[i][k][key_left][1], gold)
        if value_right[0] == float('inf'):
          table[i][j][key_right] = [min(loss_left_arc, value_right[0]), table[k+1][j][key_right][1]]
        else:
          table[i][j][key_right][0] = min(loss_left_arc, value_right[0])
          table[i][j][key_right][1] = table[k+1][j][key_right][1]

        loss_right_arc = table[i][k][key_left][0] + table[k+1][j][key_right][0] + checkGold(table[i][k][key_left][1], table[k+1][j][key_right][1], gold)
        if value_left[0] == float('inf'):
          table[i][j][key_left] = [min(loss_right_arc, value_left[0]), table[i][k][key_left][1]]
        else:
          table[i][j][key_left][0] = min(loss_right_arc, value_left[0])
          table[i][j][key_left][1] = table[i][k][key_left][1]
```

Code A.5: Exact Dynamic Oracle

```
class DynamicOracle:
  def __init__(self, parser, gold_tree):
    # Initialize the oracle with a parser and gold parse tree
    self.parser = parser
    self.gold = gold_tree
    # Track the previous action type and its cost
    self.previous_action = [-1, 0]

  def cost_left_arc(self):
    # Cost for left arc transition; infinite if illegal
    if len(self.parser.stack) < 2 or (len(self.parser.stack) == 2 and len(self.parser.buffer) != 0) or self.parser.stack[-2] == 0:
      return float('inf')

    cost = 0
    s1 = self.parser.stack[-1]
    s2 = self.parser.stack[-2]

    # Decrease cost for correct gold arc
    if self.gold[s2] == s1:
      cost -= 1

    # Check remaining gold arcs in stack and buffer
    for i in self.parser.stack + self.parser.buffer:
      if self.gold[i] == s2 or self.gold[s2] == i:
        cost += 1

    # Adjust for previous shift
    if self.previous_action[0] == 2:
      cost -= self.previous_action[1]

    return cost

  def cost_right_arc(self):
    # Cost for right arc transition; infinite if illegal
    if len(self.parser.stack) < 2 or (self.parser.stack[-2] == 0 and
   len(self.parser.buffer) != 0):
      return float('inf')

    cost = 0
    s1 = self.parser.stack[-1]
    s2 = self.parser.stack[-2]

    # Decrease cost for correct gold arc
    if self.gold[s1] == s2:
      cost -= 1

    # Check remaining gold arcs in stack and buffer
    for i in self.parser.stack + self.parser.buffer:
      if self.gold[i] == s1 or self.gold[s1] == i:
        cost += 1

    # Adjust for previous shift
    if self.previous_action[0] == 2:
      cost -= self.previous_action[1]

    return cost

  def cost_shift(self):
    # Cost for shift transition; infinite if illegal
    if len(self.parser.buffer) == 0:
      return float('inf')

    cost = 0
    s1 = self.parser.stack[-1]

    # CASE 1: s1 has a right child; right child allows costless shift
    for i in self.parser.buffer:
      if self.gold[i] == s1:
        return cost

    # CASE 2: s1 is a right child without right children
    if self.gold[s1] < s1:
      b1 = self.parser.buffer[0]
      sacrifice = 0
      orphan = False
      father = self.gold[b1]

      # Search for a lost father to create an arc with s1
      while not orphan and father != 0:
        flag = father in self.parser.stack
        if not (father in self.parser.buffer or flag):
          orphan = True
        if flag:
          return 1
        father = self.gold[father]

      # Check for lost gold arcs
      if orphan:
        return 0

      for i in self.parser.stack + self.parser.buffer:
        if self.gold[i] == b1 or self.gold[b1] == i:
          sacrifice += 1

      # Check gold arcs involving s1
      for i in self.parser.stack:
        if self.gold[i] == s1 or self.gold[s1] == i:
          cost += 1

      return min(cost, sacrifice)

    # CASE 3: s1 is a left child with no right children
    for i in self.parser.stack:
      if self.gold[i] == s1:
        cost += 1

    return cost

```

```
  def provideTransitionCosts(self):
    # Provide costs for all types of transitions
    return [self.cost_left_arc(), self.cost_right_arc(), self.
   cost_shift()]
```

Code A.6: Approximate Dynamic Oracle

```
# remove the non projective trees in the train dataset: we need to do
     this if we use a static oracle or
# if we want to train the dynamic oracle only on projective sentences
     and test it on non-projective ones to evaluate
# its the effectiveness.
train_dataset = [sample for sample in train_dataset if is_projective
    ([-1] + [int(head) for head in sample["head"]])]

# create the embedding dictionary
emb_dictionary = create_dict(train_dataset)

def process_sample(sample, get_gold_path = False):

  # put sentence and gold tree in our format
  sentence = ["<ROOT>"] + sample["tokens"]
  gold = [-1] + [int(i) for i in sample["head"]]  #heads in the gold
    tree are strings, we convert them to int

  # embedding ids of sentence words
  enc_sentence = [emb_dictionary[word] if word in emb_dictionary else
     emb_dictionary["<unk>"] for word in sentence]

  return enc_sentence, sentence, gold

def prepare_batch(batch_data):
  data = [process_sample(s) for s in batch_data]
  # sentences, paths, moves, trees are parallel arrays, each element
    refers to a sentence
  enc_sentences = [s[0] for s in data] # input_ids
  sentences = [s[1] for s in data] # sentences
  trees = [s[2] for s in data] # gold_tree
  return enc_sentences, sentences, trees

BATCH_SIZE = 32

bilstm_train_dataloader = torch.utils.data.DataLoader(train_dataset,
    batch_size=BATCH_SIZE, shuffle=True, collate_fn=partial(
    prepare_batch))
bilstm_dev_dataloader = torch.utils.data.DataLoader(dev_dataset,
    batch_size=BATCH_SIZE, shuffle=False, collate_fn=partial(
    prepare_batch))
bilstm_test_dataloader = torch.utils.data.DataLoader(test_dataset,
    batch_size=BATCH_SIZE, shuffle=False, collate_fn=partial(
    prepare_batch))
```

Code A.7: Preprocessig data

```
EMBEDDING_SIZE = 200
LSTM_SIZE = 200
LSTM_LAYERS = 2
MLP_SIZE = 200
DROPOUT = 0.2
EPOCHS = 15
LR = 0.001   # learning rate
PROBABILITY_THRESHOLD = 0.1

class BilstmParser(nn.Module):
  def __init__(self, device):
    super(BilstmParser, self).__init__()
    # Initialize the device for computations and embedding layer
    self.device = device
    self.embeddings = nn.Embedding(len(emb_dictionary),
   EMBEDDING_SIZE, padding_idx=emb_dictionary["<pad>"])

    # Initialize bi-directional LSTM
    self.lstm = nn.LSTM(EMBEDDING_SIZE, LSTM_SIZE, num_layers=
   LSTM_LAYERS, bidirectional=True, dropout=DROPOUT)

    # Initialize the feedforward layers
    self.w1 = torch.nn.Linear(8*LSTM_SIZE, MLP_SIZE, bias=True)
    self.activation = torch.nn.Tanh()
    self.w2 = torch.nn.Linear(MLP_SIZE, 3, bias=True)
    self.softmax = torch.nn.Softmax(dim=-1)

    # Initialize dropout layer
    self.dropout = torch.nn.Dropout(DROPOUT)

    # Initialize hidden state
    self.h = torch.zeros(1,1,1)

  def forward(self, x, paths, flag_enc, flag_feat):
```

```python
    # Forward pass for the model
    if flag_enc:
      # Embedding layer and dropout applied to input
      x = [self.dropout(self.embeddings(torch.tensor(i).to(self.device))) for i in x]
      # Run the bi-LSTM layer
      self.h = self.lstm_pass(x) # size(longest_sentence, batch_size, features)

    # Arrange inputs for the feedforward layers based on LSTM output and paths
    mlp_input = self.get_mlp_input(paths, self.h)

    # Run the feedforward network to get action scores
    out = self.mlp(mlp_input)

    return out

  def lstm_pass(self, x):
    # Process inputs through the LSTM
    x = torch.nn.utils.rnn.pack_sequence(x, enforce_sorted=False)
    h, (h_0, c_0) = self.lstm(x)
    h, h_sizes = torch.nn.utils.rnn.pad_packed_sequence(h) # size h: (length_sentences, batch, output_hidden_units)
    return h

  def get_mlp_input(self, configurations, h):
    # Prepare inputs for the MLP from LSTM outputs
    mlp_input = []
    zero_tensor = torch.zeros(2*LSTM_SIZE, requires_grad=False).to(self.device)
    # For each sentence in the batch
    for i in range(len(configurations)):
      # Concatenate LSTM outputs for the items in the configuration
      mlp_input.append(torch.cat([zero_tensor if configurations[i][0] == -1 else h[configurations[i][0]][i],
                                  zero_tensor if configurations[i][1] == -1 else h[configurations[i][1]][i],
                                  zero_tensor if configurations[i][2] == -1 else h[configurations[i][2]][i],
                                  zero_tensor if configurations[i][3] == -1 else h[configurations[i][3]][i]]))
    mlp_input = torch.stack(mlp_input).to(self.device)
    return mlp_input

  def mlp(self, x):
    # Feedforward network to calculate scores
    return self.softmax(self.w2(self.dropout(self.activation(self.w1(
   self.dropout(x))))))

  # Inference function to run the parser
  def infere(self, x):
    # Initialize parsers for each sentence
    parsers = [ArcStandard(i) for i in x]

    # Get embeddings for each sentence
    x = [self.embeddings(torch.tensor(i).to(self.device)) for i in x]

    # Run the LSTM
    h = self.lstm_pass(x)

    # Iterate until all sentences are parsed
    while not self.parsed_all(parsers):
      # Get current configurations and score the next moves
      configurations = self.get_configurations(parsers)
      mlp_input = self.get_mlp_input(configurations, h)
      mlp_out = self.mlp(mlp_input)
      # Take the next parsing step
      self.parse_step(parsers, mlp_out)

    # Return the predicted dependency trees
    return [parser.arcs for parser in parsers]

  def get_configurations(self, parsers):
    # Generate configurations for each parser in the batch
    configurations = []
    for parser in parsers:
      if parser.is_tree_final():
        conf = [-1, -1, -1, -1]
      else:
        if len(parser.stack) == 0:
          conf = [-1, -1, -1]
        elif len(parser.stack) == 1:
          conf = [-1, -1, parser.stack[-1]]
        elif len(parser.stack) == 2:
          conf = [-1, parser.stack[-2], parser.stack[-1]]
        else:
          conf = [parser.stack[-3], parser.stack[-2], parser.stack[-1]]
        if len(parser.buffer) == 0:
          conf.append(-1)
        else:
          conf.append(parser.buffer[0])
      configurations.append(conf)

    return configurations

  def parsed_all(self, parsers):
    # Check if all parsers have completed parsing
    for parser in parsers:
      if not parser.is_tree_final():
        return False
    return True

  # in this function we select and perform the next move according to the scores obtained.
  def parse_step(self, parsers, moves):
    moves_argm = moves.argmax(-1)
    for i in range(len(parsers)):
      if parsers[i].is_tree_final():
        continue
      else:
        if moves_argm[i] == 0:
          if parsers[i].stack[-2] != 0:
            parsers[i].left_arc()
          else:
            if len(parsers[i].buffer) > 0:
              parsers[i].shift()
            else:
              parsers[i].right_arc()
        elif moves_argm[i] == 1:
          if parsers[i].stack[-2] == 0 and len(parsers[i].buffer)>0:
            parsers[i].shift()
          else:
            parsers[i].right_arc()
        elif moves_argm[i] == 2:
          if len(parsers[i].buffer) > 0:
            parsers[i].shift()
          else:
```

```python
            if moves[i][0] > moves[i][1]:
              if parsers[i].stack[-2] != 0:
                parsers[i].left_arc()
              else:
                parsers[i].right_arc()
            else:
              parsers[i].right_arc()
```

Code A.8: BiLSTM model

```python
# Find the indices of the minimum value in nums
def find_min_indices(nums):
  min_value = min(nums)
  min_indices = [i for i, num in enumerate(nums) if num == min_value]
  return min_indices

# Execute actions for each parser in context of the approximate dynamic oracle
def execute(parsers, actions, oracles, costs):
  for parser, action, oracle, cost in zip(parsers, actions, oracles, costs):
    if parser.is_tree_final():
      continue
    else:
      if action == 0:
        parser.left_arc()
        oracle.previous_action = [0, cost[0]]
      elif action == 1:
        parser.right_arc()
        oracle.previous_action = [1, cost[1]]
      elif action == 2:
        parser.shift()
        oracle.previous_action = [2, cost[2]]

# Execute actions for each parser in context of the exact dynamic oracle
def executeExactDynamicOracle(actions, parsers, gold_trees):
  for parser, action, gold_tree in zip(parsers, actions, gold_trees):
    if parser.is_tree_final():
      continue
    else:
      if action == 0:
        if gold_tree[parser.stack[-2]] != parser.stack[-1]:
          parser.loss[-1] += 1
        parser.loss[-1] += parser.loss.pop(-2)
        parser.left_arc()

      elif action == 1:
        if gold_tree[parser.stack[-1]] != parser.stack[-2]:
          parser.loss[-2] += 1
        if len(parser.loss) > 2:
          tmp = parser.loss[-2] + parser.loss[-1]
          parser.loss[-2] = tmp
        else:
          parser.loss[0] += parser.loss[1]
        parser.loss.pop()
        parser.right_arc()

      elif action == 2:
        parser.loss.append(0)
        parser.shift()

def choose_next_amb(iteration, transition, min_cost):
  if transition in min_cost:
    return transition
  else:
    return min_cost[rd.randint(0, len(min_cost) - 1)]

def choose_next_exp(iteration, transition, min_cost):
  if iteration >= 1 and rd.random() > PROBABILITY_THRESHOLD:
    return transition
  else:
    return choose_next_amb(iteration, transition, min_cost)

def parsed_all(parsers):
  for parser in parsers:
    if not parser.is_tree_final():
      return False
  return True

def evaluate(gold, preds):
  total = 0
  correct = 0

  for g, p in zip(gold, preds):
    for i in range(1,len(g)):
      total += 1
```

```python
      if g[i] == p[i]:
        correct += 1

  return correct/total

def train(model, dataloader, criterion, optimizer, epoch, device):
  model.train()
  total_loss = 0
  count = 0
  error_count = 0

  # For each batch
  for batch in dataloader:
    # Extract sentence enconding, sentence itself and gold tree for each sentence in the batch
    enc_sentences, sentences, trees = batch
    # Reset the gradient for the current batch
    optimizer.zero_grad()
    # Containers to store transitions scores and respective gold labels
    global_transitions_scores = []
    global_gold_transitions = []

    # Flag to tell the model whether to save the encodings and the features tensor h for a new batch or not
    flag_enc = True
    # Initialize a parser and an oracle for each sentence in the batch
    parsers = [ArcStandard(s) for s in sentences]

    # CHANGE ORACLE CLASS HERE
    oracles = [DynamicOracle(parser, tree) for parser, tree in zip(parsers, trees)]

    # While each sentence hasn't been fully parsed
    while not parsed_all(parsers):

      # Save configuration: later we'll need the sequence of configurations in order to associate each one to the correct transition
      configurations = []
      for parser in parsers:
        if parser.is_tree_final():
```

```python
      configurations.append([-1, -1, -1, -1])
    else:
      if len(parser.stack) == 0:
        configurations.append([-1, -1, -1])
      elif len(parser.stack) == 1:
        configurations.append([-1, -1, parser.stack[-1]])
      elif len(parser.stack) == 2:
        configurations.append([-1, parser.stack[-2], parser.stack[-1]])
      else:
        configurations.append([parser.stack[-3], parser.stack[-2], parser.stack[-1]])
      if len(parser.buffer) == 0:
        configurations[-1].append(-1)
      else:
        configurations[-1].append(parser.buffer[0])

  # The model produce the scores for each transition given the current configuration
  transitions_scores_tensor = model(enc_sentences, configurations, flag_enc)
  transitions_scores = transitions_scores_tensor.cpu().detach().numpy()
  flag_enc = False
  # Cost of each transition for each current configuration
  costs = [oracle.provideTransitionCosts() for oracle in oracles]

  # Legal transitions for each current configuration
  legal_moves = [[index for index, value in enumerate(cost) if value != float('inf')] for cost in costs]

  # Legal transition with higher score according to the model for each current configuration
  predicted_transition = [moves[np.argmax([scores[i] for i in moves])] if not parser.is_tree_final() else -1 for scores, moves, parser in zip(transitions_scores, legal_moves, parsers)]

  # Collect the set of transitions with minimum cost for each current configuration
  min_cost_transitions = [find_min_indices(cost) for cost in costs]

  # Collect the best scoring transition among the ones with
```

```
minimum cost for each current configuration
      best_min_cost_transitions = [
        max(
        (score, i) for i, score in enumerate(scoring_quadruplet) if i in min_cost_transition
        )[1]
        for scoring_quadruplet, min_cost_transition in zip(transitions_scores, min_cost_transitions)
      ]

      # Select a transition: the one predicted by the model or a randomly chosen one from the set of minimum cost transitions
      actual_transitions = [choose_next_exp(epoch, predicted_transition[i], min_cost_transitions[i]) for i in range(len(predicted_transition))]

      # Check if the predicted transition is among the ones with minimum cost: if not we need to update the model
      for i, parser in enumerate(parsers):
        if not parser.is_tree_final():
          global_transitions_scores.append(transitions_scores_tensor[i])
          global_gold_transitions.append(best_min_cost_transitions[i])

      # Perform the decided transition
      execute(parsers, actual_transitions, oracles, costs)
      #executeExactDynamicOracle(actual_transitions, parsers, trees)

    if len(global_transitions_scores) != 0:
      global_transitions_scores = torch.stack(global_transitions_scores).to(device)
      loss = criterion(global_transitions_scores, torch.tensor(np.array(global_gold_transitions)).to(device))
      loss.backward()
      optimizer.step()
      total_loss += loss.item()
      count +=1

  return total_loss/count

def test(model, dataloader):
  model.eval()
```

```python

  gold = []
  preds = []

  for batch in dataloader:
    enc_sentences, sentences, trees = batch
    with torch.no_grad():
      pred = model.infere(enc_sentences)

      gold += trees
      preds += pred

  return evaluate(gold, preds)
```

Code A.9: Training/testing functions

```python
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Device:", device)
BiLSTM_model = BilstmParser(device)
BiLSTM_model.to(device)

criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(BiLSTM_model.parameters(), lr=LR)

# lists to store losses and scores for later being put into the
    graphs
bilstm_losses = []
bilstm_scores = []

start = time.time()

for epoch in range(EPOCHS):
  avg_train_loss = train(BiLSTM_model, bilstm_train_dataloader,
    criterion, optimizer, epoch, device)
  val_uas = test(BiLSTM_model, bilstm_dev_dataloader)

  bilstm_losses.append(avg_train_loss)
  bilstm_scores.append(val_uas)

  print("Epoch: {:3d} | avg_train_loss: {:5.3f} | dev_uas: {:5.3f} |"
    .format( epoch, avg_train_loss, val_uas))

end = time.time()
```

```

# Save the trained model weights
torch.save(BiLSTM_model.state_dict(), 'bilstm_model_weights.pth')

# minutes of training time
bilstm_training_time = (end - start)/60
```

Code A.10: Training loop

B Code II

```
import os
import numpy as np
import random as rd
import torch as T
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

!pip install datasets
!pip install conllu

import torch
import torch.nn as nn
from functools import partial
from datasets import load_dataset

!pip install evaluate

import matplotlib.pyplot as plt

!pip install "stable-baselines3[extra]>=2.0.0a4"
import gymnasium as gym
from gymnasium import spaces
from stable_baselines3.common.env_checker import check_env
from stable_baselines3.common.env_util import make_vec_env
```

Code B.1: pip and import

```python
# the function returns whether a tree is projective or not. It is
    currently
# implemented inefficiently by brute checking every pair of arcs.
def is_projective(tree):
  for i in range(len(tree)):
    if tree[i] == -1:
      continue
    left = min(i, tree[i])
    right = max(i, tree[i])

    for j in range(0, left):
      if tree[j] > left and tree[j] < right:
        return False
    for j in range(left+1, right):
      if tree[j] < left or tree[j] > right:
        return False
    for j in range(right+1, len(tree)):
      if tree[j] > left and tree[j] < right:
        return False

  return True

# the function creates a dictionary of word/index pairs: our
    embeddings vocabulary
# threshold is the minimum number of appearance for a token to be
    included in the embedding list
def create_dict(dataset, threshold=3):
  dic = {}  # dictionary of word counts
  for sample in dataset:
    for word in sample['tokens']:
      if word in dic:
        dic[word] += 1
      else:
        dic[word] = 1

  map = {}  # dictionary of word/index pairs. This is our embedding
    list
  map["<pad>"] = 0
  map["<ROOT>"] = 1
  map["<unk>"] = 2 #used for words that do not appear in our list

  next_indx = 3
  for word in dic.keys():
      if dic[word] >= threshold:
        map[word] = next_indx
        next_indx += 1

  return map

train_dataset = load_dataset('universal_dependencies', 'en_lines',
    split="train")
dev_dataset = load_dataset('universal_dependencies', 'en_lines',
    split="validation")
test_dataset = load_dataset('universal_dependencies', 'en_lines',
    split="test")

# remove the non projective trees in the train dataset
#train_dataset = [sample for sample in train_dataset if is_projective
    ([-1] + [int(head) for head in sample["head"]])]

# create the embedding dictionary
emb_dictionary = create_dict(train_dataset)

def process_sample(sample, get_gold_path = False):

  # put sentence and gold tree in our format
  sentence = ["<ROOT>"] + sample["tokens"]
  gold = [-1] + [int(i) for i in sample["head"]]  #heads in the gold
    tree are strings, we convert them to int

  # embedding ids of sentence words
  enc_sentence = [emb_dictionary[word] if word in emb_dictionary else
     emb_dictionary["<unk>"] for word in sentence]

  return enc_sentence, sentence, gold

def prepare_batch(batch_data):
  data = [process_sample(s) for s in batch_data]
  # sentences, paths, moves, trees are parallel arrays, each element
    refers to a sentence
  enc_sentences = [s[0] for s in data] # input_ids
  sentences = [s[1] for s in data] # sentences
  trees = [s[2] for s in data] # gold_tree
  return enc_sentences, sentences, trees

BATCH_SIZE = 10
```

```python

bilstm_train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, collate_fn=partial(prepare_batch))
bilstm_dev_dataloader = torch.utils.data.DataLoader(dev_dataset, batch_size=BATCH_SIZE, shuffle=False, collate_fn=partial(prepare_batch))
bilstm_test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False, collate_fn=partial(prepare_batch))
```

Code B.2: dataset management

```python
class PrioritizedReplayBuffer():
  def __init__(self, max_size, input_shape, alpha=0.9):
    """
    Initialize the Prioritized Replay Buffer.

    Args:
        max_size (int): The maximum size of the buffer.
        input_shape (tuple): The shape of the inputs.
        alpha (float): Determines how much prioritization is used, with 0 corresponding to no prioritization.
    """
    self.mem_size = max_size
    self.mem_cntr = 0
    self.alpha = alpha  # The exponent alpha determines how much prioritization is used

    # Initialize memory for states, actions, rewards, terminal flags, and priorities
    self.state_memory = np.zeros((self.mem_size, *input_shape), dtype=np.float32)
    self.new_state_memory = np.zeros((self.mem_size, *input_shape), dtype=np.float32)
    self.action_memory = np.zeros(self.mem_size, dtype=np.int64)
    self.reward_memory = np.zeros(self.mem_size, dtype=np.float32)
    self.terminal_memory = np.zeros(self.mem_size, dtype=np.uint8)
    self.priority_memory = np.zeros(self.mem_size, dtype=np.float32) + 1e-5  # Initialize with small positive values
    self.max_priority = 1.0  # Initial max priority

  def store_transition(self, state, action, reward, state_, done):
    """
    Store a transition in the buffer.

    Args:
        state: The state of the environment before the action.
        action: The action taken.
        reward: The reward received.
        state_: The state of the environment after the action.
        done: Whether the episode has ended.
    """
    index = self.mem_cntr % self.mem_size  # Circular buffer

    # Store the transition in the respective memory arrays
    self.state_memory[index] = state
    self.new_state_memory[index] = state_
    self.action_memory[index] = action
    self.reward_memory[index] = reward
    self.terminal_memory[index] = done

    # Assign the max priority seen so far to new experiences
    self.priority_memory[index] = self.max_priority

    self.mem_cntr += 1

  def sample_buffer(self, batch_size, beta=0.5):
    """
    Sample a batch of transitions from the buffer.

    Args:
        batch_size (int): The size of the batch to sample.
        beta (float): The exponent for adjusting the importance-sampling weights.

    Returns:
        Tuple containing states, actions, rewards, next states, terminals, indices of the sampled transitions, and the importance-sampling weights.
    """
    # Determine the range of memory to sample from
    num_sampled_elements = min(self.mem_cntr, self.mem_size)
    priorities = self.priority_memory[:num_sampled_elements]

    # Normalize priorities and convert to probabilities
    scaled_priorities = np.power(priorities, self.alpha)
```

```python
        sample_probs = scaled_priorities / np.sum(scaled_priorities)

        # Sample experiences based on probabilities
        chosen_indices = np.random.choice(num_sampled_elements, batch_size, replace=False, p=sample_probs)

        # Retrieve sampled experiences
        states = self.state_memory[chosen_indices]
        actions = self.action_memory[chosen_indices]
        rewards = self.reward_memory[chosen_indices]
        states_ = self.new_state_memory[chosen_indices]
        terminal = self.terminal_memory[chosen_indices]

        # Compute importance-sampling weights and adjust with beta
        weights = np.power(self.mem_size * sample_probs[chosen_indices], -beta)
        weights /= np.max(weights)  # Normalize for stability
        weights = torch.tensor(weights, dtype=torch.float32).view(-1, 1)  # Convert to tensor and reshape

        return states, actions, rewards, states_, terminal, chosen_indices, weights

    def update_priorities(self, indices, priorities):
        """
        Update the priorities of the sampled transitions in a vectorized manner.

        Args:
            indices (list or numpy.ndarray): Indices of the sampled transitions.
            priorities (list or numpy.ndarray): New priorities for the sampled transitions.
        """
        # Ensure that no priority is set to exactly 0 by using np.maximum, as a priority of 0 would mean a transition is never sampled.
        # Adding a small value (1e-5) ensures all priorities are non-zero and transitions have a chance of being sampled.
        priorities = np.maximum(priorities, 1e-5)

        # Update the priorities in a vectorized manner.
        # This is generally faster and more efficient than a loop, especially for large arrays.
        self.priority_memory[indices] = priorities

        # Update the maximum priority with the largest priority in the new set.
        # This value is used to set the priority for new experiences (ensuring they have a high chance of being sampled initially).
        self.max_priority = max(self.max_priority, np.max(priorities))
```

Code B.3: Proportional Prioritized Replay Buffer

```python
class DuelingDeepQNetwork(nn.Module):
  def __init__(self, lr, n_actions, name, input_dims, chkpt_dir):
    """
    Initialize the Dueling Deep Q Network.

    Args:
      lr (float): Learning rate for the optimizer.
      n_actions (int): Number of possible actions.
      name (str): Name of the network, used in saving and loading models.
      input_dims (tuple): Dimensions of the input state.
      chkpt_dir (str): Directory where the checkpoints (model weights) are saved.
    """
    super(DuelingDeepQNetwork, self).__init__()
    self.chkpt_dir = chkpt_dir
    self.checkpoint_file = os.path.join(self.chkpt_dir, name)

    # Define the first fully connected layer
    self.fc1 = nn.Linear(*input_dims, 128)
    # Define the layer for estimating the state-value function V
    self.V = nn.Linear(128, 1)
    # Define the layer for estimating the advantage function A
    self.A = nn.Linear(128, n_actions)

    # Set up the optimizer (Adam) and the loss function (Mean Squared Error)
    self.optimizer = optim.Adam(self.parameters(), lr=lr)
    self.loss = nn.MSELoss()
    # Define the device (use GPU if available)
    self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')
    self.to(self.device)

  def forward(self, state):
    """
    Perform a forward pass through the network.

    Args:
      state (torch.Tensor): The input state.

    Returns:
      V (torch.Tensor): The estimated state-value function.
      A (torch.Tensor): The estimated advantage function.
    """
    flat1 = F.relu(self.fc1(state))  # Pass the state through the first fully connected layer
    V = self.V(flat1)  # Compute the state-value function
    A = self.A(flat1)  # Compute the advantage function

    return V, A

  def save_checkpoint(self):
    """
    Save the model's current state.
    """
    print('...saving checkpoint...')
    T.save(self.state_dict(), self.checkpoint_file)

  def load_checkpoint(self):
    """
    Load the model's state from a saved checkpoint.
    """
    print('...loading checkpoint...')
    self.load_state_dict(T.load(self.checkpoint_file))
```

Code B.4: Dueling Deep Q-Network

```
class Agent():
  def __init__(self, gamma, epsilon, lr, n_actions, input_dims, mem_size, batch_size, eps_min=0.01, eps_dec=5e-7, replace=1000, beta_start=0.5, beta_increment_per_sampling=0.001, beta_max=1.0, chkpt_dir='tmp/dueling_ddqn'):
    """
    Initialize the agent with given hyperparameters and network parameters.

    Args:
        gamma (float): discount factor for future rewards.
        epsilon (float): initial exploration rate for epsilon-greedy action selection.
        lr (float): learning rate for updating the neural network.
        n_actions (int): number of possible actions the agent can take.
        input_dims (tuple): dimensions of the input features.
        mem_size (int): size of the replay memory.
        batch_size (int): number of experiences sampled from memory for each learning step.
        eps_min (float): minimum value for epsilon (exploration rate).
        eps_dec (float): decrement value for epsilon after each episode.
        replace (int): number of steps after which the target network weights are updated.
        beta_start (float): initial value of beta for importance-sampling weights.
        beta_increment_per_sampling (float): increment value for beta after each sampling.
        beta_max (float): maximum value for beta.
        chkpt_dir (str): directory where model checkpoints are saved.
    """
    # Initialize parameters
    self.gamma = gamma
    self.epsilon = epsilon
    self.lr = lr
    self.n_actions = n_actions
    self.input_dims = input_dims
    self.batch_size = batch_size
    self.eps_min = eps_min
    self.eps_dec = eps_dec
    self.replace_target_cnt = replace
    self.beta = beta_start
    self.beta_increment_per_sampling = beta_increment_per_sampling
    self.beta_max = beta_max
    self.chkpt_dir = chkpt_dir
    self.learn_step_counter = 0
    self.action_space = [i for i in range(self.n_actions)]

    # Initialize memory and Dueling DQNs for current and target network
    self.memory = PrioritizedReplayBuffer(mem_size, input_dims)
    self.q_eval = DuelingDeepQNetwork(lr, n_actions, 'q_eval', input_dims, chkpt_dir)
    self.q_next = DuelingDeepQNetwork(lr, n_actions, 'q_next', input_dims, chkpt_dir)

    # Initialize variables for averaging network weights
    self.average_q_eval_state_dict = None  # To store the averaged state dict of the Q_eval network
    self.networks_counter = 0  # To count the number of networks added to the average

  def choose_action(self, observation):
    """
    Choose an action based on the current state and the epsilon-greedy policy.

    Args:
      observation (np.array): the current state observation.

    Returns:
      action (int): the action chosen by the agent.
    """
    if np.random.random() > self.epsilon:
      # Exploitation: choose the best action according to the network's output
      state = T.tensor(np.array(observation), dtype=T.float32).to(self.q_eval.device)
      _, advantage = self.q_eval.forward(state)
      action = T.argmax(advantage).item()
    else:
      # Exploration: choose a random action
      action = np.random.choice(self.action_space)
    return action

  def store_transition(self, state, action, reward, state_, done):
    """
    Store a transition in the replay buffer.

    Args:
      state (np.array): the starting state.
      action (int): the action taken.
      reward (float): the reward received.
      state_ (np.array): the next state after taking the action.
      done (bool): whether the episode is finished.
    """
```

```python
    self.memory.store_transition(state, action, reward, state_, done)

  def replace_target_network(self):
    """
    Update the target network by copying the weights from the evaluation network.
    This happens every 'replace_target_cnt' learning steps.
    """
    if self.learn_step_counter % self.replace_target_cnt == 0:
      self.q_next.load_state_dict(self.q_eval.state_dict())

  def update_average_network(self, current_state_dict):
    """
    Update the running average of the Q_eval network weights.
    This is intended to stabilize the training by smoothing out the variations in the network weights over training steps.

    Args:
      current_state_dict (dict): state_dict of the current Q_eval network.
    """
    self.networks_counter += 1
    if self.average_q_eval_state_dict is None:
      self.average_q_eval_state_dict = {k: v.clone().detach() for k, v in current_state_dict.items()}
    else:
      new_average_q_eval_state_dict = {}
      for key in self.average_q_eval_state_dict.keys():
        new_average_q_eval_state_dict[key] = (
            self.average_q_eval_state_dict[key] * (self.networks_counter - 1)
            + current_state_dict[key]
        ) / self.networks_counter
      self.average_q_eval_state_dict = new_average_q_eval_state_dict

  def decrement_epsilon(self):
    """
    Decrement the epsilon value to reduce exploration over time.
    """
    self.epsilon = max(self.epsilon - self.eps_dec, self.eps_min)

  def save_models(self):
    """
```

```python
        Save the current and target network models.
        """
        self.q_eval.save_checkpoint()
        self.q_next.save_checkpoint()

    def load_models(self):
        """
        Load the saved models for the current and target networks.
        """
        self.q_eval.load_checkpoint()
        self.q_next.load_checkpoint()

    def learn(self):
        """
        The learning process for the agent. Samples a batch of
      experiences and updates the network.
        """
        if self.memory.mem_cntr < self.batch_size:
            return  # Do not learn until enough samples are available

        self.q_eval.optimizer.zero_grad()

        # Update the target network and the average network at the
      specified intervals
        if self.learn_step_counter % self.replace_target_cnt == 0:
            self.replace_target_network()
            self.update_average_network(self.q_eval.state_dict())

        # Sample a batch from the replay buffer
        states, actions, rewards, states_, dones, indices, weights = self
      .memory.sample_buffer(self.batch_size, self.beta)

        states = T.tensor(states).to(self.q_eval.device)
        actions = T.tensor(actions).to(self.q_eval.device)
        dones = T.tensor(dones).to(self.q_eval.device)
        rewards = T.tensor(rewards).to(self.q_eval.device)
        states_ = T.tensor(states_).to(self.q_eval.device)
        weights = weights.clone().detach().requires_grad_(True).to(self.
      q_eval.device)

        batch_indices = np.arange(self.batch_size)

        # Load the averaged network weights for predicting the next Q-
```

```
values
        self.q_eval.load_state_dict(self.average_q_eval_state_dict)

        V_s, A_s = self.q_eval.forward(states)
        V_s_avg, A_s_avg = self.q_eval.forward(states_)

        q_pred = T.add(V_s, (A_s - A_s.mean(dim=1, keepdim=True)))[
batch_indices, actions]
        q_next = T.add(V_s_avg, (A_s_avg - A_s_avg.mean(dim=1, keepdim=
True)))
        q_next[dones.bool()] = 0.0  # Set Q value of next state to 0 if
the episode ended
        q_target = rewards + self.gamma * q_next[batch_indices, T.argmax(
A_s_avg, dim=1)]

        # Compute loss, perform backpropagation, and update network
weights
        loss = self.q_eval.loss(q_target, q_pred) * weights  # Apply
importance-sampling weights
        loss = loss.mean()  # Average the loss over the batch
        loss.backward()
        self.q_eval.optimizer.step()

        # Update learning step counter and epsilon
        self.learn_step_counter += 1
        self.decrement_epsilon()

        # Increment beta, ensuring it doesn't exceed beta_max
        self.beta = min(self.beta + self.beta_increment_per_sampling,
self.beta_max)

        # Update the priorities in the replay buffer based on TD error
        td_errors = (q_target - q_pred).detach().cpu().numpy()
        new_priorities = np.abs(td_errors) + 1e-5  # Ensure priorities
are non-zero
        self.memory.update_priorities(indices, new_priorities)
```

Code B.5: Agent

```
class ArcStandard:
    def __init__(self, sentence, tree):
        self.gold_tree = tree
        self.sentence = sentence
        self.buffer = [i for i in range(len(self.sentence))]
        self.stack = []
        self.arcs = [-1 for _ in range(len(self.sentence))]
        self.prev_actions = [None, None, None, None, None]

        # three shift moves to initialize the stack
        self.shift()
        self.shift()
        if len(self.sentence) > 2:
          self.shift()

        self.loss = [0 for i in range(len(self.stack))]

    def shift(self):
        b1 = self.buffer[0]
        self.buffer = self.buffer[1:]
        self.stack.append(b1)
        if len(self.prev_actions) == 5:
          self.prev_actions.pop(0)
        self.prev_actions.append('shift')

    def left_arc(self):
        o1 = self.stack.pop()
        o2 = self.stack.pop()
        self.arcs[o2] = o1
        self.stack.append(o1)
        if len(self.prev_actions) == 5:
          self.prev_actions.pop(0)
        self.prev_actions.append('left_arc')
        if len(self.stack) < 2 and len(self.buffer) > 0:
          self.shift()
          if len(self.prev_actions) == 5:
            self.prev_actions.pop(0)
          self.prev_actions.append('shift')

    def right_arc(self):
        o1 = self.stack.pop()
        o2 = self.stack.pop()
        self.arcs[o1] = o2
        self.stack.append(o2)
        if len(self.prev_actions) == 5:
          self.prev_actions.pop(0)
        self.prev_actions.append('right_arc')
        if len(self.stack) < 2 and len(self.buffer) > 0:
```

```python
      self.shift()
      if len(self.prev_actions) == 5:
        self.prev_actions.pop(0)
      self.prev_actions.append('shift')

  def is_tree_final(self):
    return len(self.stack) == 1 and len(self.buffer) == 0

  def print_configuration(self):
    s = [self.sentence[i] for i in self.stack]
    b = [self.sentence[i] for i in self.buffer]
    print(s, b)
    print(self.arcs)

  def get_valid_actions(self):
    """
    Determine the valid actions that can be taken from the current state of the parser.

    Returns:
      list: A list of valid actions.
    """
    valid_actions = ['shift', 'left_arc', 'right_arc']

    # 'shift' is not valid if the buffer is empty
    if len(self.buffer) == 0:
      valid_actions.remove('shift')

    # 'left_arc' is not valid if:
    # 1. The stack has less than 2 elements
    # 2. The stack has exactly 2 elements but the buffer is not empty
    # 3. The second-to-last element on the stack is the root (0)
    if len(self.stack) < 2 or (len(self.stack) == 2 and len(self.buffer) != 0) or self.stack[-2] == 0:
      valid_actions.remove('left_arc')

    # 'right_arc' is not valid if:
    # 1. The stack has less than 2 elements
    # 2. The second-to-last element on the stack is the root (0) and the buffer is not empty
    if len(self.stack) < 2 or (self.stack[-2] == 0 and len(self.buffer) != 0):
      valid_actions.remove('right_arc')

    return valid_actions

  def get_binary_features(self, N=10):
    """
    Construct the binary feature vector for the current state.
    Each of the top 10 tokens from the stack and the first 10 tokens
   from the buffer
    will have their gold head position binary encoded using 5 bits,
   and additional bits
    indicating if the gold head is lost and if all dependents are
   already collected.

    Args:
      N (int): The number of tokens from the stack and buffer to
   consider.

    Returns:
      np.array: The binary feature vector representing the current
   state.
    """
    # Initialize the binary feature vector
    binary_features = []

    # Get the top N tokens from the stack and the first N tokens from
     the buffer
    stack_elements = self.stack[-N:] if len(self.stack) >= N else
   self.stack + [-1] * (N - len(self.stack))
    buffer_elements = self.buffer[:N] if len(self.buffer) >= N else
   self.buffer + [-1] * (N - len(self.buffer))

    # Combine stack and buffer elements for easier indexing
    combined_elements = stack_elements + buffer_elements

    # Get the complete stack and buffer for checking if the gold head
     is lost
    complete_elements = self.stack + self.buffer

    # Encode the position of the gold head for each element in
   stack_elements and buffer_elements
    for token_index in combined_elements:
      if token_index == -1:
        binary_features.extend([-1, -1, -1, -1, -1, -1, -1])  #
   Padding representation with 7 bits
      else:
        if token_index == 0:
          binary_features.extend([1, 1, 1, 1, 1, 0] + [self.
   has_collected_all_dependents(token_index)])
        else:
          gold_head = self.gold_tree[token_index]
          gold_head_pos = combined_elements.index(gold_head) if
   gold_head in combined_elements else -1
          gold_head_lost = 1 if (self.gold_tree[token_index] not in
   complete_elements and token_index != 0) else 0
          all_dependents_collected = self.
   has_collected_all_dependents(token_index)
          if gold_head_pos == -1:
            binary_features.extend([-1, -1, -1, -1, -1,
   gold_head_lost, all_dependents_collected])  # Gold head is lost or
    not in the 20 elements
          else:
            binary_features.extend([int(bit) for bit in self.
   position_to_binary(gold_head_pos)] + [gold_head_lost,
   all_dependents_collected])
    # Encode the last 5 (or fewer, with padding) actions leading to
   this state
    binary_features.extend(self.get_padded_prev_actions(self.
   prev_actions))

    # Encode all valid actions in this state
    # Assuming 'get_valid_actions' returns a list of valid actions in
     the current state
    valid_actions = self.get_valid_actions()
    binary_features.extend([1 if action in valid_actions else 0 for
   action in ['shift', 'left_arc', 'right_arc']])

    return np.array(binary_features)

  def position_to_binary(self, pos, max_pos=20):
    """
    Convert a position to a 5-bit binary representation.
    If the position is out of range (lost or not among the 20
   elements), return '00000'.

    Args:
        pos (int): The position to be converted.
```

```python
        max_pos (int): The maximum position value (20 for top 10 in stack and first 10 in buffer).

    Returns:
        str: A 5-bit binary string representing the position.
    """
    if pos < 0 or pos >= max_pos:
        return '00000'
    return format(pos, '05b')

  def has_collected_all_dependents(self, first_common_parent):
    for token in self.stack:
      if self.gold_tree[token] == first_common_parent:
        return 0

    for token in self.buffer:
      if self.gold_tree[token] == first_common_parent:
        return 0

    return 1

  def action_to_binary(self, action):
    """
    Convert an action to its binary (one-hot encoded) representation.

    Args:
      action (str): The action to be converted.

    Returns:
      list: The binary representation of the action.
    """
    if action == 'left_arc':
      return [1, 0]
    elif action == 'right_arc':
      return [0, 1]
    elif action == 'shift':
      return [1, 1]
    else:  # For padding or unknown actions
      return [0, 0]

  def get_padded_prev_actions(self, prev_actions, max_prev_actions=5):
    """
    Get the binary representations of previous actions, padded with zeros if there are fewer than 'max_prev_actions'.

    Args:
      prev_actions (list): The list of the last few actions taken.
      max_prev_actions (int): The maximum number of previous actions to consider.

    Returns:
      list: A flattened list containing the binary representations of previous actions, padded with zeros.
    """
    # Convert each previous action to its binary representation
    binary_prev_actions = [self.action_to_binary(action) for action in prev_actions]

    # Calculate the number of actions to pad
    num_padding = max_prev_actions - len(binary_prev_actions)

    # Pad with vectors representing 'no action'
    binary_prev_actions.extend([self.action_to_binary(None)] * num_padding)

    # Flatten the list of binary vectors into a single list
    return [bit for action_bits in binary_prev_actions for bit in action_bits]
```

Code B.6: Modified Arc-Standard

```
class DependencyParsingEnv(gym.Env):
  metadata = {'render.modes': ['human']}

  def __init__(self, sentence, tree, max_steps_per_episode=5):
    super(DependencyParsingEnv, self).__init__()
    self.sentence = sentence
    self.tree = tree
    self.parser = ArcStandard(sentence, tree)
    self.previous_action = [-1, 0]
    self.positive_reward = 1
    self.current_step = 0
    self.max_steps_per_episode = max_steps_per_episode

    # Define action and observation space
    self.action_space = spaces.Discrete(3)
    self.observation_space = spaces.Box(low=-1, high=1, shape=(self.
   parser.get_binary_features().shape[0],), dtype=np.float32)

  def get_valid_actions(self):
    valid_actions = self.parser.get_valid_actions()
    valid_actions_indexes = []
    if 'left_arc' in valid_actions:
      valid_actions_indexes.append(0)
    if 'right_arc' in valid_actions:
      valid_actions_indexes.append(1)
    if 'shift' in valid_actions:
      valid_actions_indexes.append(2)

    return valid_actions_indexes

  def step(self, action):
    self.current_step += 1
    valid_actions = self.get_valid_actions()
    # Map the action to the parser's functions
    if action == 0 and action in valid_actions:  # left_arc
      self.parser.left_arc()
    elif action == 1 and action in valid_actions:  # right_arc
      self.parser.right_arc()
    elif action == 2 and action in valid_actions:  # shift
      self.parser.shift()

    # Compute the reward for the current action
    reward, _ = self.computeReward(self.parser.stack, self.parser.
   buffer, self.parser.gold_tree, action, self.previous_action)

    # Update the previous action
    self.previous_action = [action, reward]

    # Check if the episode (parsing of one sentence) is done
    done = self.parser.is_tree_final()

    # Check if max steps per episode is reached
    truncated = False
    if self.current_step >= self.max_steps_per_episode:
      done = True
      truncated = True

    # Get the next state representation
```

```python
    state = self.parser.get_binary_features().astype(np.float32)

    # Additional info can be added if necessary
    info = {}

    return state, reward, done, truncated, info

  def reset(self, seed=None, options=None):
    # Reset the state of the environment to an initial state
    self.parser = ArcStandard(self.sentence, self.tree)
    self.previous_action = [-1, 0]
    self.current_step = 0
    observation = self.parser.get_binary_features().astype(np.float32)
    info = {}  # Optional: can contain additional information
    return observation, info

  def render(self, mode='human', close=False):
    # Render the environment to the screen
    self.parser.print_configuration()

  def computeReward(self, stack, buffer, gold_tree, action, previous_action):
    # LEFT_ARC
    if action == 0:
      if len(stack) < 2 or (len(stack) == 2 and len(buffer) != 0) or stack[-2] == 0:
        return -100, False
      reward = 0
      s1 = stack[-1]
      s2 = stack[-2]

      if gold_tree[s2] == s1:
        reward += 2

      for i in stack:
        if gold_tree[i] == s2 or gold_tree[s2] == i:
          reward -= 1

      for i in buffer:
        if gold_tree[i] == s2 or gold_tree[s2] == i:
          reward -= 1

      if previous_action[0] == 2:
        reward -= previous_action[1]

      if reward == 1:
        reward = self.positive_reward

      return reward, False
    # RIGHT_ARC
    elif action == 1:
      if len(stack) < 2 or (stack[-2] == 0 and len(buffer) > 0):
        return -100, False
      reward = 0

      s1 = stack[-1]
      s2 = stack[-2]

      if gold_tree[s1] == s2:
        reward += 2

      for i in stack:
        if gold_tree[i] == s1 or gold_tree[s1] == i:
          reward -= 1

      for i in buffer:
        if gold_tree[i] == s1 or gold_tree[s1] == i:
          reward -= 1

      if previous_action[0] == 2:
        reward -= previous_action[1]

      if reward == 1:
        reward = self.positive_reward

      return reward, False
    # SHIFT
    elif action == 2:
      if len(buffer) == 0:
        return -100, False

      reward = 0
      s1 = stack[-1]

      for i in buffer:
```

```
      if gold_tree[i] == s1:
        return self.positive_reward, False # a right child allows a costless shift

    # s1 is a right child without right children
    if gold_tree[s1] < s1:
      b1 = buffer[0]
      sacrifice = 0
      # search for a lost father so that we can create an arc between s1 and the orphan node
      orphan = False
      father = gold_tree[b1]
      while not orphan and father != 0:
        flag = father in stack
        if (father not in buffer and not flag):
          orphan = True
        if flag:
          return -1, False
        father = gold_tree[father]

      if orphan:
        return 0, False

      for i in stack:
        if gold_tree[i] == b1 or gold_tree[b1] == i:
          sacrifice -= 1
      for i in buffer:
        if gold_tree[i] == b1 or gold_tree[b1] == i:
          sacrifice -= 1

      for i in stack:
        if gold_tree[i] == s1 or gold_tree[s1] == i:
          reward -= 1

      if reward == 0:
        return self.positive_reward, False

      return max(reward, sacrifice), False

    # s1 is a left child with no right children
    for i in stack:
      if gold_tree[i] == s1:
        reward -= 1

    if reward == 0:
      reward = self.positive_reward
    return reward, False
```

Code B.7: Dependency parsing environment

```
def evaluateSingleTree(gold, preds):
  total = 0
  correct = 0

  for i in range(1,len(gold)):
    total += 1
    if gold[i] == preds[i]:
      correct += 1

  return correct/total

max_forked_episodes = 5
max_episode_length = 100
max_epochs= 30

agent = Agent(gamma=0.9, epsilon=1.0, lr=5e-4, n_actions=3, input_dims=[153], mem_size=50000, batch_size=1000, eps_min=0.01, eps_dec=1e-7, replace=100)

for epoch in range(max_epochs):
  for batch_data in bilstm_train_dataloader:
    enc_sentences, sentences, trees = batch_data
    for enc_sentence, sentence, tree in zip(enc_sentences, sentences, trees):
      for _ in range(max_forked_episodes):
        env = DependencyParsingEnv(sentence, tree, max_steps_per_episode=max_episode_length)
        state = env.reset()[0]
        for _ in range(max_episode_length):
          action = agent.choose_action(state)
          valid_actions = env.get_valid_actions()
          if np.random.rand() < 0.05 and len(valid_actions) != 0:  # Forking probability
            action = np.random.choice(valid_actions)
          next_state, reward, done, truncated, _ = env.step(action)
          agent.store_transition(state, action, reward, next_state, done)
          agent.learn()
          state = next_state
          if done or truncated:
            break
        if done:
          break  # No more forking if the true end of the sentence is reached

  count = 0
  tot_loss = 0
  for batch_data in bilstm_dev_dataloader:
    enc_sentences, sentences, trees = batch_data
    for enc_sentence, sentence, tree in zip(enc_sentences, sentences, trees):
      env = DependencyParsingEnv(sentence, tree, max_steps_per_episode=500)
      state = env.reset()[0]
      while not env.parser.is_tree_final():
        action = agent.choose_action(state)
        next_state, reward, done, truncated, _ = env.step(action)
        state = next_state
        if done or truncated or reward == -100:
          count += 1
          tot_loss = evaluateSingleTree(tree, env.parser.arcs)
          break

  # Print epoch summary
  print(f'Epoch: {epoch}, UAS: {tot_loss/count}')
```

Code B.8: DQN Training loop

References

[1] Oron Anschel, Nir Baram, and Nahum Shimkin. *Averaged-DQN: Variance Reduction and Stabilization for Deep Reinforcement Learning*. 2017. arXiv: 1611.01929 [cs.AI].

[2] Lauriane Aufrant, Guillaume Wisniewski, and François Yvon. "Exploiting Dynamic Oracles to Train Projective Dependency Parsers on Non-Projective Trees". In: *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Ed. by ACL. ACL. New Orleans, United States, June 2018, pp. 413–419. URL: https://hal.science/hal-01813394.

[3] Miguel Ballesteros and Joakim Nivre. "Going to the Roots of Dependency Parsing". In: *Computational Linguistics* 39 (Mar. 2013), pp. 5–13. DOI: 10.1162/COLI_a_00132.

[4] Jason Eisner and G. Satta. "Efficient Parsing for Bilexical Context-Free Grammars and Head Automaton Grammars". In: *Annual Meeting of the Association for Computational Linguistics*. 1999. URL: https://api.semanticscholar.org/CorpusID:333410.

[5] Yoav Goldberg and Joakim Nivre. "A Dynamic Oracle for Arc-Eager Dependency Parsing". In: Dec. 2012, pp. 959–976.

[6] Yoav Goldberg and Joakim Nivre. "Training Deterministic Parsers with Non-Deterministic Oracles". In: *Transactions of the Association for Computational Linguistics* 1 (Oct. 2013), pp. 403–414. ISSN: 2307-387X. DOI: 10.1162/tacl_a_00237. eprint: https://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00237/1566681/tacl_a_00237.pdf. URL: https://doi.org/10.1162/tacl%5C_a%5C_00237.

[7] Dan Jurafsky and James H. Martin. *Speech and language processing : an introduction to natural language processing, computational linguistics, and speech recognition*. Upper Saddle River, N.J.: Pearson Prentice Hall, 2009. ISBN: 9780131873216 0131873210. URL: `http://www.amazon.com/Speech-Language-Processing-2nd-Edition/dp/0131873210/ref=pd_bxgy_b_img_y`.

[8] Eliyahu Kiperwasser and Yoav Goldberg. *Simple and Accurate Dependency Parsing Using Bidirectional LSTM Feature Representations*. 2016. arXiv: `1603.04351 [cs.CL]`.

[9] Tom Schaul et al. *Prioritized Experience Replay*. 2016. arXiv: `1511.05952 [cs.LG]`.

[10] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: `http://incompleteideas.net/book/the-book-2nd.html`.

[11] Ziyu Wang et al. *Dueling Network Architectures for Deep Reinforcement Learning*. 2016. arXiv: `1511.06581 [cs.LG]`.

[12] Xiang Yu, Thang Vu, and Jonas Kuhn. "Approximate Dynamic Oracle for Dependency Parsing with Reinforcement Learning". In: Jan. 2018, pp. 183–191. DOI: `10.18653/v1/W18-6021`.

Acknowledgments

Chi mi conosce bene sa che amo essere laconico: poche parole chiare e senza fronzoli. Tuttavia, al termine di questo percorso, sento la necessità di esprimere un profondo riconoscimento verso chi mi ha accompagnato finora, facendomi crescere come persona e come studente.

Comincio con un profondo ringraziamento a Giorgio Satta, che non solo ha saputo farmi da Virgilio durante questo lungo lavoro di tesi, ma che mi ha anche guidato e consigliato per ciò che sarà il futuro, aiutandomi a dare forma concreta alle mie vaghe aspirazioni. In lui ho trovato una passione coinvolgente per il campo del NLP, che ora condivido, ed un vero e proprio mentore capace di camminare sia davanti a me per guidarmi nella propria area di conoscenza, sia al mio fianco nell'esplorare argomenti a lui nuovi. Grazie per la pazienza, la disponibilità ed il sostegno che mi ha dimostrato.

Un ringraziamento speciale va anche a Gian Antonio Susto e a Niccolò Turcato, che si sono prestati ad un progetto non propriamente nel loro campo di ricerca e, ciononostante, si sono sempre interessati al suo sviluppo con contributi non indifferenti. Grazie per il tempo prezioso che mi avete dedicato.

Ringrazio Giacomo, Paolo e Marco con cui ho affrontato buona parte del mio percorso universitario condividendo fatiche e successi: siete stati compagni di viaggio insostituibili ed indispensabili. Vi auguro il meglio per i vostri percorsi, ovunque vi portino.

Una calorosa riconoscenza va anche a Giuliano: un amico e un fratello che ha saputo risollevarmi ed incoraggiarmi nei momenti più duri. Avere accanto una persona del suo calibro è un onore e una fonte d'ispirazione più unica che rara.

Infine un ultimo ringraziamento va alla mia famiglia: dai miei genitori, Cinzia e Paolo, che mi hanno permesso di studiare fino ad ora senza farmi mai mancare nulla, alla nonna Lindia, pilastro portante fin dalla mia nascita.

Ringrazio anche quelli che purtroppo oggi non ci sono più: il nonno Igino e gli zii Luana e Luciano mi hanno cresciuto come un figlio e non posso che essere loro grato. Ultimi, ma non meno importanti, Elisa e Marco: grazie della vostra compagnia, vedervi percorrere la vostra strada è uno dei miei più grandi privilegi.

Pietro