ARC-FACTORED BIAFFINE DEPENDENCY PARSING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF LINGUISTICS
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Timothy Dozat
May 2019

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christopher Manning, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Dan Jurafsky

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Martin Kay

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

This thesis describes a simple approach to neural arc-factored dependency parsing, building on neural machine learning techniques that have gained considerable popularity in recent years. Dependency parsing is a way of identifying the latent syntactic and semantic relationships between words in a sentence, with solid foundations in linguistic theory that I describe in some detail. In this work, I introduce new classification techniques that extend the affine softmax classifier ubiquitous in machine learning that would otherwise be inappropriate for parsing. What's more, I demonstrate that the new biaffine classification techniques can be derived mathematically from the same principles that yield the affine softmax classifier. Related works either use an alternative to the proposed biaffine classifiers—based on feedforward neural attention—or else use an entirely different parsing algorithm—known as transition-based parsing—based on constituency parsing. In this work, I find evidence that the biaffine classifiers outperform the traditional attention-based classifiers, and that the arc-factored system outperforms transition-based parsers more broadly. I also demonstrate that the hyperparameter choices are optimal or near optimal, with significant deviations either leading to overfitting or underfitting. Consequently, any modifications to the architecture that yield better accuracy are unlikely to be due to simply compensating for poor hyperparameters. The basic system can be batched to parse large documents very quickly, and achieves accuracy comparable to state-of-the-art on the most popular English benchmark. However, the original system makes a few design choices that introduce complications for other languages, namely a reliance on whole word tokens and part-of-speech tags. To solve the first limitation, I have the system construct word representations from characters, so that the model can learn how morphology expressed through orthography reflects syntactic structure. To solve the second, I minimally adapt the architecture of the parser so it can be trained as a sequence labeler. A tagger that directly uses insights gleaned from the parser can be trained on any dependency treebank with gold part-of-speech tags. This approach achieved the highest performance at tagging and parsing on the 2017 CoNLL shared task on dependency parsing, inspiring most of the top-performing systems of the 2018 shared task. I also extend the system for multitask tagging, such that morphological features and language-specific part-of-speech tags are conditioned on the predicted coarse-grained universal tag. Finally, I modify the edge classifier to condition predictions directly on the relative location of words, so the system

can more effectively leverage linearization and distance. Both of these make statistically significant improvements to accuracy. In order to accommodate dependency formalisms that don't adhere to strict tree structures, I minimally adapt the parser once more to produce arbitrary dependency graphs instead of dependency trees. I again ablate the system to explore how important the different hyperparameters and components of the system are, finding that while most of them do make a statistically significant difference, in general the differences are very small and the system is very robust. The work in this thesis not only contributes narrowly to the field of dependency parsing, but also more broadly provides tools for tasks with more complex dependencies than sequence labeling or classification.

Contents

List of Tables

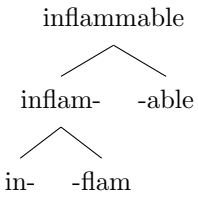
List of Figures

Chapter 1

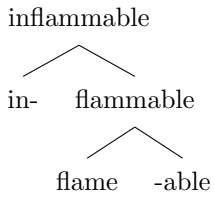
Introduction

This thesis describes a relatively simple, high-accuracy, and theoretically-motivated neural machine learning architecture that can be trained to identify the hierarchical linguistic structure of any sentence in a given language. Linguistic structure comes in many forms, some of which are *flat*—with no discernable substructures—and others of which are *hierarchical*—composed of smaller components, which themselves are composed of smaller components, potentially unboundedly. One familiar kind of flat structure is word segmentation (or, in the realm of Natural Language Processing, tokenization). While sentences are written with spaces, spoken language generally doesn't contain pauses between words; a spoken utterance is more or less one continuous stream of sound. On the other hand, an intuitive kind of latent hierarchical structure comes in the form of affixation; one can add a prefix or suffix to a word to change that word's meaning, and then one can add another prefix or suffix to change it again, and so on. This can be seen in the two opposing meanings of the word *inflammable*, which has both a prefix and a suffix. When the prefix is applied lower in the hierarchical structure (i.e. "first"), as in Ex. (1.1a), one gets the positive meaning; but when the suffix is applied higher (i.e. "second"), one gets the negative one.

(1.1) a. [in[flam]]mable
'The object can catch on fire'



b. in[[flam]mable]
'The object cannot catch on fire'



In Ex. (1.1a), the prefix *in-* (here meaning 'into', or 'on') combines with the Latinate root *flam-*, to create the word *inflame*, which historically meant 'to catch on fire' (cf. *inflammation*). Then, the suffix *-able* combines with the word *inflame*, creating a new word that means 'able to catch

on fire'. However, the word has been reanalyzed since its inception to have a different hierarchical morphological structure, provided in Ex. (1.1b). The root *flam-* on its own has taken the meaning of 'to catch on fire', combining with *-able* first, creating a word that means 'able to catch on fire'. The new word can then be negated by the homophonous prefix *in-* (this time meaning 'not'), creating a word meaning 'not able to catch on fire'. This structure is latent because again, there are no explicit cues to indicate how the roots and affixes are organized.

The kind of latent structure that this thesis aims to model is known as *syntax*. Syntax can be more or less described as characterizing the latent hierarchical linguistic structure that organizes words into valid, meaningful sentences. Of course, it's impossible to fully separate syntax from other aspects of language, such as morphology, semantics, and pragmatics. One of the primary functions of syntactic structure is to organize words into sentences that convey semantic meanings from one individual to another. In Ex. (1.2a), *Sandy* is the indvidual experiencing the *wanting*, and in Ex. (1.2b)—which exchanges the two individuals—*Kim* is. Ex. (1.2c) uses the same words but in a scrambled order, and as such does not convey any information; by hypothesis, this is because it lacks a syntactic structure that could be generated by Standard American English.

(1.2) a. Sandy wants to hug Kim.
b. Kim wants to hug Sandy.
c. * Kim Sandy wants hug to.

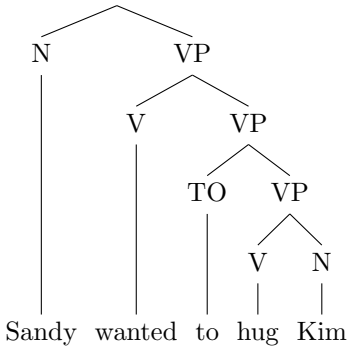
While tokenization is largely recoverable from orthography in many languages, syntactic structure is not. Moreover, while virtually all linguists agree that syntactic structure exists in some form, it is difficult even for native speakers or trained linguists to agree upon a language's exact syntactic rules and structures. Thus identifying syntactic structures can be very difficult; however, the close relationship between syntax and semantics makes the syntactic structure of a sentence a valuable asset to natural language understanding pipelines.

The syntax of a sentence is closely related to its meaning, so having an analysis of the syntactic structure of a sentence is useful for practical applications that deal with natural language understanding in some way. The primary focus of this research is on using neural machine learning to automatically generate these hierarchical syntactic analyses according to some predefined linguistic framework. Before the rise of neural machine learning techniques, syntactic structure was assumed to be necessary for a wide variety of these kinds of tasks. Even in the current age of neural networks, knowledge base population and relation extraction—both of which involve identifying simple facts in a text or series of texts—are places where syntactic structure is still being used very successfully (Chaganty et al., 2017; Zhang et al., 2018b), and neural architectures that take advantage of hierarchical structure are gradually being found to outperform those that don't (He et al., 2017; Strubell et al., 2018). But as mentioned above, there are multiple competing theories of syntax that all make incompatible assumptions, each with its own strengths and weaknesses and several of which actually posit multiple distinct layers of hierarchy. Thus it's advantageous to develop a general-purpose

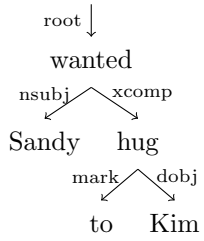
system that can learn the syntactic structures proposed by any of a variety of different theories. To this end, this thesis describes a machine learning architecture that can be trained to generate any linguistic representation that can be reduced to a *dependency graph*. This is in opposition to a *dependency tree* and a *constituency tree*, which differ in a few critical ways. A constituency tree is similar to the kind of linguistic structure described in Ex. (1.1): just as how contiguous morphemes combine together to form logical units, contiguous spans of words combine together to form phrases; and furthermore, the order in which the main word (the *head* word) combines with its modifiers is often meaningful. In a dependency *tree*, by contrast, the words being grouped into a unit do not need to be contiguous, and there is no hierarchy among a word's dependents (the modifiers that are subordinate to it). A dependency graph is an extension of a dependency tree that allows one word or phrase to be hierarchically subordinate to multiple (or no) other word or phrases. These three types of structure are shown in Ex. (1.3).

(1.3) Sandy wanted to hug Kim.

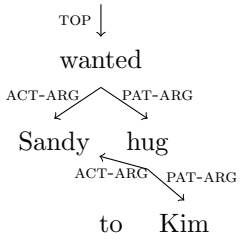
a.



b.



c.



The exact details of these representations will be described later; the key point is that they all group the words together in different ways. Working bottom-up, the constituency representation in Ex. (1.3a) first groups the verb *hug* and the noun *Kim* into one unit, which then gets assigned the label VP, for *verb phrase*. A larger VP is then created by grouping the smaller VP with the word *to*, which then gets grouped into an even larger VP when it combines with *wanted*. In the dependency tree in Ex. (1.3b), all of a word's dependents are identified and grouped with that word simultaneously, and their relationships to that word are all labeled. So first *hug* combines with its auxiliary marker *to* and its direct object *Kim*, then *wanted* combines with its nominal subject *Sandy* and its verbal complement *hug*. In the dependency graph (Ex. 1.3c), *Sandy* is grouped with both *wanted* and *hug*, while *to* isn't grouped with any other words. The dependency representations don't (and in fact generally *can't*) stipulate hierarchy between the subject and complement verb. Instead, they use different labels to formally differentiate the relationship between a verb and these two kinds of modifiers, while maintaining a flat structure. Put succinctly, constituency trees identify (unlabeled) relationships between adjacent labeled phrases, and dependency trees identify labeled relationships between (potentially nonadjacent) words. Then, dependency trees require each word

to modify exactly one other word (except the one at the top of the hierarchical tree); dependency graphs relax this restriction, so that a word can be subordinate to no other words or multiple other words.

While the difference between dependency trees and dependency graphs might seem trivial, it has substantial implications for parsing. There are two popular parsing paradigms for dependency representations—*transition-based* and *arc-factored* (also known as *graph based*, but this thesis will prefer the former term to avoid confusion with graph-structured and tree-structured formalisms). The transition-based paradigm adapts an efficient constituency tree parsing algorithm, modifying it to produce dependency trees instead (Nivre, 2003). It implicitly prunes away unlikely subtrees, so that its simplest incarnation has a very efficient, $O(n)$ time complexity; however, it can only produce trees that mimic the constraints of constituency trees, with strict adjacency requirements. Relaxing the adjacency requirements requires foregoing the efficient theoretical complexity bounds, though empirically the runtime does not suffer drastically (Nivre, 2009). Relaxing the tree requirement, so that words can depend on more or fewer than one head, requires substantially more complexity (Wang et al., 2018). The arc-factored paradigm makes no attempt to prune away improbable edges, examining each pair of words to see if one depends on the other (McDonald et al., 2005). If the dependency formalism is unconstrained—with no tree or adjacency requirements—then the system's predictions can be used without post-processing. Alternatively, if the formalism has tree constraints or adjacency constraints, a spanning tree algorithm can be used to impose them. Historically, comparing two words used to be very computationally expensive, so arc-factored parsers—which need to make $O(n^2)$ feature-based comparisons—have been considerably less popular than transition-based parsers, which make $O(n)$ comparisons. However, one advantage of neural networks is that there are many opportunities for parallelization, so that doing n^2 operations isn't actually much slower than doing n operations. In fact, the pruning strategy of transition-based algorithms inhibits efficient parallelization, meaning that arc-factored parsers have the potential to be faster than transition-based ones for large documents of text. Not only does the pruning of transition-based approaches now inhibit efficiency, but the system runs the risk of pruning away correct subtrees prematurely, with workarounds to this substantially increasing engineering complexity. This suggests that arc-factored approaches to dependency parsing have the potential to be both faster and more accurate than comparable transition-based approaches.

With these considerations in mind, in the work at the crux of this thesis—Dozat and Manning (2017)—I propose a relatively straightforward neural architecture for producing dependency trees. Following the conventional neural techniques for natural language processing, the system uses vector-space embeddings for the words and part-of-speech tags in a sentence as input to a recurrent neural network. Recurrent neural networks represent a convenient and popular way to contextualize the representation of each word based on the rest of sentence. Each word's recurrent vector is then put through one feedforward layer—the most basic kind of neural network—to generate a "head"

vector-space representation for that word, and another feedforward layer to generate a "dependent" representation. This split-representation approach is motivated by the asymmetric relationship between head and dependent; that is, the features useful for finding a word's dependents may not be the same as the features useful for finding a word's head.

There are a number of ways that one can define a function to score a pair of words based on their head and dependent representations. One approach, based off the attention mechanism proposed by Bahdanau et al. (2014), would involve concatenating the representations and feeding them into a multi-layer perceptron with a single output. Another approach, which is similar to work by Luong et al. (2015), would take the dot product of the two representations, and yet another (drawing inspiration from the same authors) would involve feeding them into a bilinear transformation with one output. This work proves mathematically that the probability that the system is trying to model—the probability of an edge between two words, given the two feature representations—can be rewritten into a variation on the bilinear transformation that includes linear terms as well (making it bi*affine*). An analogous approach is optimal for predicting the labels for each edge. Empirical comparison of the proposed biaffine approach and approach related to Bahdanau et al.'s (2014) attention (which several other researchers employ; cf. Zhang et al. 2017; Kiperwasser and Goldberg 2016) suggests that the theoretically-motivated approach outperforms the more *ad hoc* one. This work also takes the position that thorough ablation studies and hyperparameter tuning are critical for making any scientific claims, so it additionally compares the base system against even simpler variants—finding that they underperform to varying degrees—and reports notable hyperparameter settings. The resulting system is very fast and achieves very good accuracy, outperforming the other arc-factored systems and approaching the performance of the more complex state-of-the-art transition-based parser.

The primary benchmark used to evaluate parser performance at the time of the original work was a treebank of English. While that work did evaluate performance on treebanks of other languages, the other systems being compared against only reported on a subset of these treebanks. This makes it difficult to make claims about the efficacy of the different approaches in different situations, such as when the training dataset is much smaller or the language has more flexible word order. The 2017 and 2018 CoNLL shared tasks (Zeman et al., 2017, 2018) provided a testbed to compare different approaches to tagging and parsing (as well as other NLP tasks) for a wide variety of languages. The 2017 shared task included test data for 81 treebanks from 49 languages, and the 2018 shared task included test data for 82 treebanks from 57 languages. This allowed different parsers to be compared along a wide variety of different dimensions.

In my contribution to Stanford's submission to the shared tasks (Dozat et al., 2017; Qi et al., 2018), I extend the basic parser in a number of ways in order to accommodate some of the many different linguistic properties and data conventions. Part-of-speech tagging is a critical pre-processing step for many dependency parsers, including the one proposed in here. In order to ensure that

every language had access to a high-quality tagger, the first addition in the 2017 submission is a trainable part-of-speech tagger with similar architecture to the parser. This allows the tagger to take advantage of the insights into hyperparameters gleaned from the original work. Additionally, the basic parser treated each word as atomic, making it difficult for the system to learn relationships between words with similar orthography. For example, the English words *want* and *wants* both have similar meanings and syntactic properties; however, in the original system, for simplicity there was no mechanism in place to explicitly capture the similarity between them. In English treebanks this likely doesn't affect performance too much, but for languages with richer inflectional morphology, it represents a much bigger cause for concern. To address this limitation, the system constructs representations for words using their individual character sequences.

Both the part-of-speech tagger and the dependency tree parser achieved the highest performance of any team at the 2017 competition; ablation studies and per-language comparisons with other teams' submissions reveal that the gains come from a number of sources. Paying careful attention to part-of-speech tagging and character-level orthography made significant improvements to the parser, validating the extra complexity associated with modeling these features. Additionally, the arc-factored strategy outperformed the transition-based baseline by a wider margin on languages with more crossing dependencies, suggesting that the arc-factored approach to parsing may be more effective at parsing languages where word order is less predictive of grammatical functions. The system also performed particularly well on languages with more training data, suggesting that the hyperparameters were well-tuned for high-resource circumstances but may need to be better optimized for low-resource ones.

Nearly all top-performing submissions to the 2018 competition drew inspiration from Stanford's 2017 submission: of the top ten systems, only one of them did not use a biaffine arc-factored architecture for parsing. Instead of making changes to the system architecture, the highest-performing submission (Che et al., 2018) ensembled multiple copies of Stanford's 2017 submission and utilized ELMo embeddings (Peters et al., 2018), which leverage information from large unlabeled corpora more effectively than the traditional word embeddings used in Stanford's submissions. I likewise made a few more augmentations to the version of the system that Stanford submitted to the 2018 competition, though I put more emphasis on neural architecture.

In the shared task, the part-of-speech tagger needs to label each word in the sentence according to three different labeling schemes. However, some of the labels are highly correlated; for example, it would generally be unreasonable to classify a word as a verb in one of the labeling schemes and a noun in the other. Thus the tagger in the 2018 version of the system conditions predictions for the harder (i.e. higher-entropy) tagging schemes on the easiest (i.e. lowest-entropy) one. That is, first it labels the words according to the tagset with the fewest options to choose from, then it labels each word according to the other tagsets, taking into consideration the first tag it predicted for that word. This approach is shown to both improve performance on the harder tagsets and also slightly

improve the consistency of the tags.

Furthermore, the basic version of the parser doesn't explicitly condition head assignments on the relative locations of each word. It bilinearly transforms the neural features between each pair of words into a score. The consequence of this is that any information about the order of or distance between a word and its head must be extracted from the recurrent neural network alone, and can only be used in a polynomial function. It's not clear that this is the most efficient way to take advantage of these features; one might worry that this hurts performance on longer sentences when there are multiple words with the right syntactic and semantic properties to be a given word's head, but are too far away or on the wrong side. This is especially concerning for languages with more relaxed word order, where there are no discernable syntactic rules that dictate attachment, and where one would like to take advantage of a soft "attach to the nearest possible candidate" heuristic. To address this concern, the system enhances the scorer with a way to explicitly condition head predictions on word order and distance. This is then shown to make a significant impact on performance, with the effect being carried by longer sentences.

The parser described so far was designed to produce dependency *trees*, rather than less restricted dependency *graphs*, which allow each word to depend on multiple or no other words. While transition-based approaches struggle to efficiently and accurately generate these representations, I show that the neural architecture explored in this thesis can be adapted to handle these graphs very easily (Dozat and Manning, 2018). Instead of optimizing for *softmax* cross-entropy, to find the single best head for a given word, the system optimizes for *sigmoid* cross-entropy, allowing it to find *all* the best heads for a given word. Labeling each head-dependent relation can then be done in the exact same way as in the tree-structure parser. One problem that arises is that the loss for the unlabeled parser—the part that makes head predictions—is significantly smaller than the loss from the labeler, so the optimizer focuses too much on getting the correct labels and not enough on getting the correct heads. This is addressed by upweighting the unlabeled parser loss to balance it out with the label loss, allowing the system to efficiently optimize both the head-predicting and edge-labeling subtasks. As is standard in this thesis, particular care is taken to ensure that the additional complexity incurred by low-level architecture enhancements is in fact empirically justified. In doing so, it is demonstrated that unlabeled parsing and labeling can be done simultaneously in one step without reducing accuracy, suggesting that the system can be simplified even further.

Overall, the main goal of this thesis is to propose a relatively simple neural dependency parser, along with a few useful extensions to improve accuracy under certain circumstances (such as for languages other than English or non-tree structured dependency formalisms). It aims to do this in a way that rigorously tests the individual components of the system to justify any complexity beyond what is strictly necessary theoretically or empirically. The core architectural innovations—biaffine attention and biaffine classifiers—are likewise rigorously motivated theoretically, being a natural extension of the standard affine classifier ubiquitous elsewhere in machine learning. On a

lower level, this thesis introduces and motivates new neural components not specific to dependency parsing, utilizing biaffine transformations in a wide variety of different classifiers that can be used for making predictions when the information comes from two different sources. The impact of the parser in this thesis is multi-fold: it is fast and accurate, making it good for use as a black-box in downstream tasks (Strubell et al., 2018); it provides a highly accurate baseline against which to test alternative architectures (Ma et al., 2018); similarly, because of the simple and straightforward architecture of the parser, it represents an easy-to-extend starting point for additional research on dependency parsing (Wang et al., 2017); finally, because it doesn't rely on any task-specific architecture, it can be used as an auxiliary task in multitask systems (Shi and Zhang, 2017; Clark et al., 2018).

The rest of this thesis is laid out as follows. Chapter 2 describes the theoretical origins of a number of popular dependency frameworks, illuminating the relationship between academic linguistic theory and its application in computationally useful representations. Chapter 3 goes over the machine learning background assumed in the rest of the thesis, and proves that biaffine attention follows from the same principles that yield affine classifiers. Chapter 4 surveys the parsing literature, describing in detail the most prominent and relevant approaches to parsing (both transition-based and arc-factored). Chapter 5 describes the basic architecture of the parser, originally published as Dozat and Manning (2017). Chapter 6 describes the augmentations made in the 2017 and 2018 CoNLL shared tasks to handle phenomena in non-English languages, originally published as Dozat et al. (2017); Qi et al. (2018). Chapter 7 describes the extension of the parser to graph-structured dependency formalisms, originally published as Dozat and Manning (2018). Finally, Chapter 8 concludes the thesis.

Chapter 2

Syntax: From Theory to Practice

2.1 Introduction

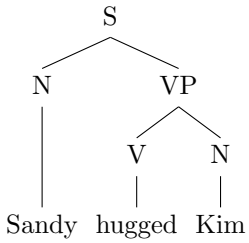
The simplest approach to many tasks in the realm of natural language processing (NLP) is the "bag-of-words" approach, which makes use of the tokens (and perhaps bigrams) in the sentence while ignoring the order they appeared in. For text classification objectives, this approach can yield relatively good performance (Wang and Manning, 2012). In modern neural network systems, this normally translates to a bag-of-embeddings approach, where each word is assigned a learned vector representation (Jin et al., 2016). However, not only is this approach insufficient for many more complex tasks (such as those involving natural language generation), but even within text classification the "bag-of-embeddings" model is normally outclassed by more complex neural models that retain the original order of words in the sentence (Tang et al., 2015; Lai et al., 2015). Similarly, some researchers have found that neural systems that only take advantage of word order can be surpassed by systems that utilize higher-level linguistic information (Tai et al., 2015; Marcheggiani and Titov, 2017). In addition to preserving the original order of words in the sentence, each word can be labeled with additional details relating to features such as its *part of speech* (such as whether a word is a noun, verb, adjective, etc.) or *named entity* status (whether a word is part of a name, organization, location, etc.). These features are limited to describing the sequence of independent words in a sentence—however, sentences are known to have hierarchical, *syntactic* structure, which individual word labels are generally unable to capture.

The goal of this chapter is to describe some of the major theories of syntax as well as the phenomena that motivated them, but focusing on the ones that have been used as inspiration for computer-friendly formalisms. This thesis is centered around a computational system for generating analyzing any arbitrary string of text into these formalisms, so it's important to have an understanding not only of their empirical utility in downstream applications but also of their theoretical origins and soundness. That is, the syntactic formalisms that this system generates analyses for

are not arbitrary or *ad hoc*; they derive from decades of (often competing) theoretical research on language—simplified to be practical to use, read, and annotate—which this chapter aims to outline.

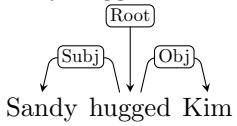
The main goal of the syntax branch of linguistic theory is to describe cross-linguistically general mechanisms that can be parameterized for a given language to generate all and only the acceptable sentences in that language. It sometimes includes the goal of describing how meaning can be assigned to a parsed sentence in the language. While different theories of syntax make different assumptions about nature of these mechanisms, essentially all of them acknowledge that words are recursively grouped into syntactic units, or *constituents*. For theories with multiple distinct levels of representation, the level that aims to model this recursive hierarchy is known as the level of *constituency structure*. Many popular syntactic theories require that these constituents be contiguous and labeled with abstract syntactic categories; these contiguous labeled groups are known as *phrase structures*, and together they form a sentence's *phrase structure tree*. For example, in the relatively theory-neutral phrase structure tree in Ex. (2.1), the verb (labeled V) and its nominal object (N) are grouped into a *verb phrase* (VP), which in turn is grouped with the subject into a complete *sentence* (S). Phrasal labels such as VP and S are also known as *syntactic categories*.

(2.1) Sandy hugged Kim.



Other syntactic theories assume that constituents can be discontiguous. Instead of encoding syntactic hierarchy between words and phrases, they encode the hierarchy between pairs of individual words. Less prominent words are made to *depend* on more prominent words, with the most important word marked as the *root* of the hierarchy. Each word is labeled with the relationship it has with its *head* word, the word it depends on. This kind of hierarchical representation is known as a *dependency tree*.

(2.2) Sandy hugged Kim.



The remainder of this section will provide evidence for syntactic structure in language and introduce the reader to a few linguistic phenomena that will help illustrate the mechanisms used by different syntactic theories. It will also characterize some of the theoretical strengths and weaknesses of phrase structure and dependency approaches to modeling surface-level syntax. Subsequent

sections will describe the linguistic theories that have been used as inspiration for general-purpose computational representations of sentences, focusing on differences in the underlying syntactic representations they posit. The discussion of each new theory will emphasize phenomena that it's particularly well-equipped to handle or that motivated key aspects of it. In some cases though, these phenomena are complex and require laborious effort from experts to annotate accurately. In these cases, they may not always be represented in formalisms inspired by these theories. Still, an understanding of the challenges that motivated the theories will illuminate some of the oddities of the formalisms and make related tools more accessible. It will also show the reader what aspects of language have yet to be represented in formalisms that have gained widespread popularity.

2.1.1 Constituency structure

There is strong empirical evidence in favor of constituency structure in human language (Pāṇini and Katre, 1987; Chomsky, 1957), though phrase structure is not universally accepted (Tesnière, 1959). Some constructions involve dislocating or stranding specific groups of words, suggesting that these groups form a logical unit. *Verb phrase topicalization*—shown in Ex. (2.3), with "*" indicating that a sentence is unacceptable—is one such construction.

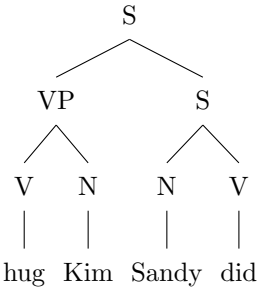
(2.3) a. Sandy wanted to *hug Kim*, and Sandy did *hug Kim*.

b. Sandy wanted to *hug Kim*, and *hug Kim* Sandy did.

c. * Sandy wanted to *hug Kim*, and *hug* Sandy did *Kim*.

d. * Sandy wanted to *hug Kim*, and Sandy *hug Kim* did.

e. * Sandy wanted to *hug Kim*, and *Kim hug* Sandy did.



The only difference between the acceptable sentence (Ex. 2.3b) and the unacceptable sentence (Ex. 2.3c) is whether or not the object is dislocated along with the verb. Furthermore, arbitrary words from the conjoined clause cannot be topicalized—the subject cannot be topicalized alongside the verb, shown in Ex. (2.3d). As indicated by the difference between Ex. (2.3b) and Ex. (2.3e), the rule in English that the object must follow the verb holds in topicalized sentences as well as in non-topicalized ones. Similarly, a nonfinite verb and its object can be stranded in answer to a question, but only if they follow the same linear precedence constraint that orders the verb before the object in basic clauses.

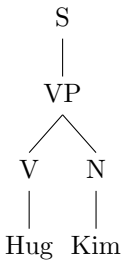
(2.4) What did Sandy want to do?

a. Sandy wanted to *hug Kim*.

b. *Hug Kim*.

c. * Sandy *hug Kim*.

d. * *Kim hug*.



As with verb phrase topicalization, the verb and its object can occur in a stranded answer (Ex. 2.4b), but the subject cannot (Ex. 2.4c) and the verb and its object must occur in the usual order (Ex. 2.4d). The uniformity between basic sentences, VP-topicalized sentences, and stranded answers can be generalized by making two assumptions: (a) that a nonfinite verb and its object are grouped into a single unit that follows certain constraints no matter where it is in the sentence; and (b) that verb phrase topicalization and stranded answers involve moving or dislocating this unit.

There is normally one word or phrase inside a larger phrase with special status, known as the *head* word, which defines how the larger phrase behaves. Often (though not universally), this word or phrase must be present in order for the larger phrase to exist. For example, in a phrase like *hug Kim*, the verb *hug* is generally assumed to be the head, because the object is optional (cf. *Sandy and Kim hugged*), and the phrase can occur in most of the same places that intransitive verbs without an object can occur.

(2.5) Sandy left.

(2.6) a. Sandy wanted to *leave*, and *leave* Sandy did.

b. * Sandy wanted to *leave*, and Sandy *leave* did.

(2.7) What did Sandy want to do?

a. Sandy wanted to *leave*.

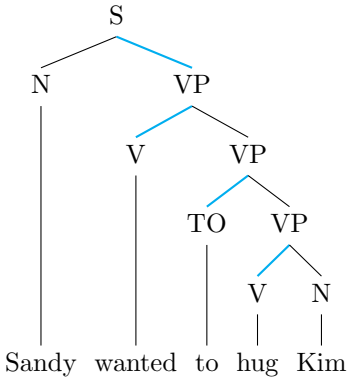
b. Leave.

c. * Sandy leave.

Because phrases composed of a transitive verb and its object can appear in the same constructions as a lone intransitive verb, it can be inferred that the verb is more "important" than its object for determining where the phrase can appear. Many researchers have observed that *function words*, such as prepositions and complementizers like *if* and *that*, are normally heads (Jackendoff, 1977;

Hudson, 1984; Chomsky, 1986). Many (but not all) theories include articles here as well. The head words and phrases of the sentence *Sandy wanted to hug Kim* are marked with thick colored edges in Ex. (2.8).

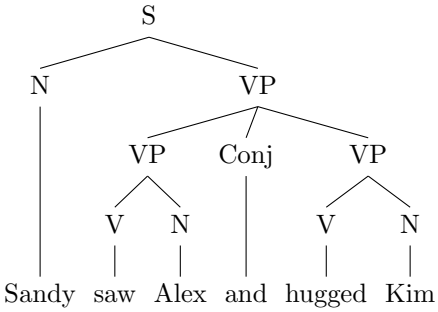
(2.8) Sandy wanted to hug Kim



Verbs taking infinitive complements (as in Ex. 2.8) is a linguistic phenomenon known as *control*; it's fairly common in English but introduces some complexities that need to be accounted for, so this sentence will serve as an example throughout this chapter.

All syntactic theories assume in one way or another that the hierarchical structure of a sentence is not arbitrary—rather, it's related to how the meaning of the sentence is constructed (or divided up). By hypothesis, the larger phrase unit combines together the individual meanings of its constituents to create a larger, coherent unit of meaning. This is sometimes referred to as *semantic compositionality*. For example, in the sentence in Ex. (2.9), the words *saw Alex* form a verb phrase, as do *hugged Kim*.

(2.9) Sandy saw Alex and hugged Kim.



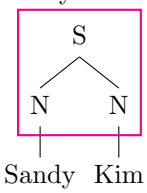
Furthermore, these verb phrases provide information about the events being described. *Alex* is the person being seen, not the person being hugged, and *Kim* is described as the person being hugged, not seen. The conjunction then combines the two verb phrases so that any relationships higher in the constituency hierarchy apply to both events. This can be seen when the conjoined verb phrase combines with the noun *Sandy*, making the person *Sandy* both the *seer* and *hugger*.

2.1.2 Phrase structure grammars vs dependency grammars

Effectively all theories of syntax distinguish between constituency structure—detailed above—and some kind of abstract structure, althogh in modern Transformational Grammar (known as Minimalism; Chomsky 1994) the line is somewhat blurred. Sections 2.2 through 2.6 will describe differences between the underlying representations that different theories posit, which are all very different. This section will discuss the strengths and weaknesses of the two primary approaches to representing constituency structure: phrase structure and dependency structure.

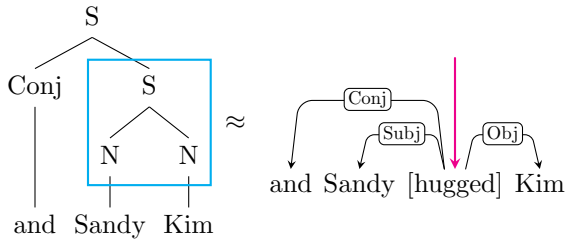
In theories that assume phrase structure, headedness has to be imposed on the constituency structures through independent mechanisms. That is, nothing inherently blocks constituency trees like Ex. (2.10).

(2.10) Sandy Kim



Phrase structure grammarians normally address this by assuming some variant of X-bar theory (Chomsky, 1970; Jackendoff, 1977), which places restrictions on what phrase structure rules are allowed in the theory to simulate headedness. Proponents of theories that assume dependency structure, on the other hand, point out that dependency trees are *built* on the notion of headedness. A phrase structure like Ex. (2.10)—where neither word is the head, and the phrase behaves like a finite clause—has no analog in a dependency framework. Some proponents of phrase structure would counter that this is actually a desirable characteristic, allowing for constructions where there truly isn't evidence for an overt head (Ginzburg and Sag, 2000). A dependency grammar could not provide a structure for Ex. (2.11) without inserting a silent copy of the main verb in the second clause. Not only does this introduce parsing complications, but it turns out that empty tokens in dependency parses are only required for this one specific type of ellipsis, rendering the analysis somewhat unsatisfying.

(2.11) Alex hugged Sandy, and Sandy, Kim



Phrase structure grammars thus need to stipulate an independent mechanism that explains why most phrases have heads, whereas dependency grammars need to stipulate an independent mechanism to

explain why some phrases seem to lack them.

Phrase structure trees are particularly useful for capturing more fine-grained word order generalizations common in languages like English (which was the primary language of study when the academic popularity of syntax took off). One instance of this relates to modifiers in English (Sadler and Arnold, 1994). In English, single-word adjectives typically occur to the left of the noun they modify. However, when an adjective takes an argument or a prepositional phrase modifier, the adjective phrase (AP) normally has to come to the right of the noun. Native speaker intiution dictates that in order for the adjective phrase to appear before the noun, it has to be "hyphenated", meaning that it syntactically and intonationally behaves as a single word.

(2.12) a. A worthy man.
b. * A man worthy.
(2.13) a. A man worthy of praise.
b. * A worthy of praise man.
(2.14) a. A man green with envy.
b. * A green with envy man.
(2.15) a. A worthy-of-praise man.
b. A green-with-envy man.

Peculiarly, adjectives modified with adverbial intensifiers can still occur to the left of the noun, provided that they have no following prepositional phrase modifiers or arguments. Intensified adjectives occurring after the noun are generally unacceptable, although they may be used to evoke an archaic tone.

(2.16) a. A very worthy man.
b. * A man very worthy.
(2.17) a. A man very worthy of praise.
b. * A very worthy of praise man.
c. A very worthy-of-praise man.

Adjectival and adverbial modification in theories that assume context-free phrase structure (for instance, X-bar theory, assumed in the following trees; Jackendoff 1977) normally involves rules of the sort in Ex. (2.18), where the head category (here marked with an asterisk) is grouped with the modifier to create a phrase with the same category as the original head.

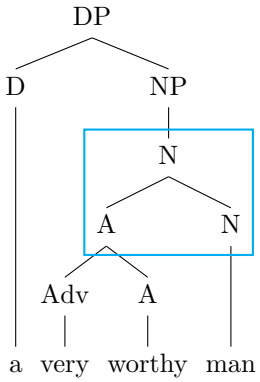
(2.18) XP $\rightarrow$ XP* ModP

This is known as an *adjunction* rule, because the modifying phrase ModP joins with the head phrase XP but doesn't fundamentally chage it—the whole unit is still an XP and can appear wherever the unmodified version can appear. The linearization of adjectives relative to nouns in English can be explained succinctly in a phrase structure grammar by stipulating that there are two levels of syntactic category: the "word" level and the "phrase" level. Intensifier adverbs (of category Adv) adjoin to adjectives leftwards at the word level (A), whereas prepositional phrases (PP) adjoin

rightwards at the phrase level (AP). Since adjunction is assumed to create the same syntactic category as the head (Ex. 2.18), an Adv adjoining to an A would create another A. The intensified adjective would then have the same distribution as a single-word adjective. This means that pattern of adjectives modifying nouns can be described similarly: a noun N can only be modified by an A to the left (Ex. 2.19), and a noun phrase NP can only be modified by an AP to the right (Ex. 2.20). The context-free phrase structure rules in Ex. (2.19–2.20) formalize the pattern just described, with asterisks indicating the head of the phrase.

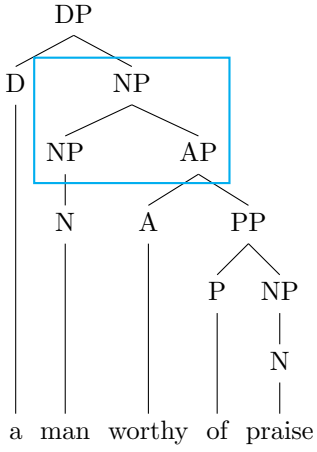
(2.19) a. $A \rightarrow Adv\ A^*$
$N \rightarrow A\ N^*$

b. A very worthy man.



(2.20) a. $AP \rightarrow AP^*\ PP$
$NP \rightarrow NP^*\ AP$

b. A man worthy of praise.

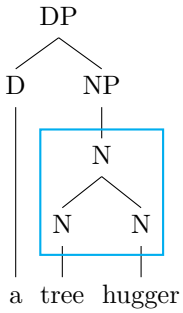


One striking observation is that in both pairs of rules, word-level adjunction happens to the left and phrase-level adjunction happens to the right. This raises the question: does that observation hold elsewhere in English? One place where it would seem to is noun compounding. Nouns can be placed in a single phrase side-by-side, in which case the noun to the left is considered the modifier

(Ex. 2.21), or one can be embedded in a prepositional phrase, in which case the modifying phrase goes to the right (Ex. 2.22).

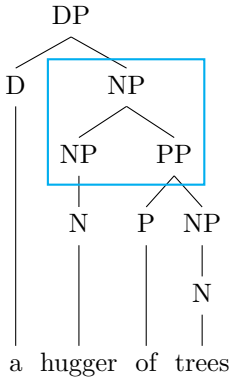
(2.21) a. N → N N*

b. A tree hugger.



(2.22) a. NP → NP* PP

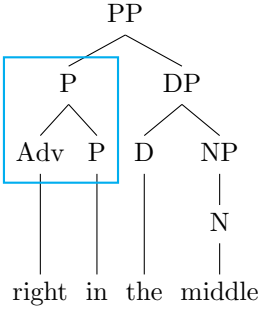
b. A hugger of trees.



Prepositions and adverbs, like adjectives, can be intensified. This is normally done with the intensifier *right*, occuring to the left (Ex. 2.23, 2.24). Some adverbs can be modified with prepositional phrases; in this case, the modifying phrase goes to the right (Ex. 2.25).

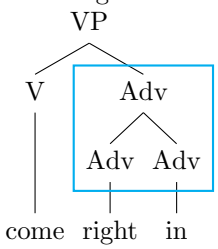
(2.23) a. P → Adv P*

b. Right in the middle.



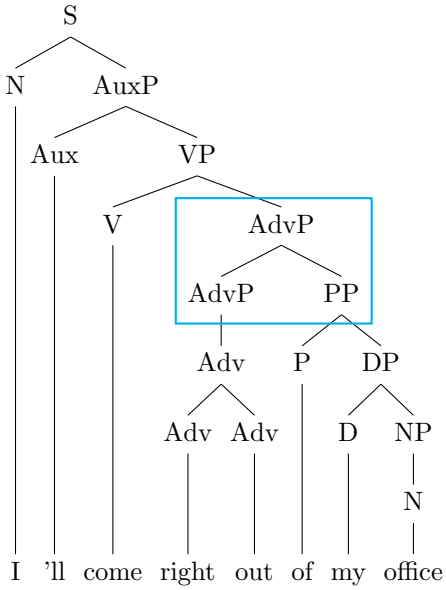
(2.24) a. Adv → Adv Adv*

b. Come right in.



(2.25) a. Adv → Adv* PP

b. I'll come right out of my office.

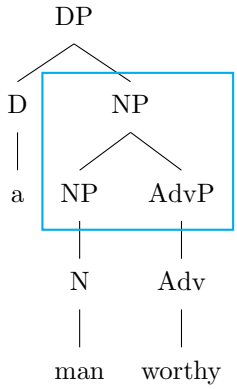


This generalization can be captured in a phrase structure grammar with only two reasonable stipulations: adjunction at the word level is head-final, but adjunction at the phrase level is head-initial (Ex. 2.26); and unary branching phrases should be avoided when possible. The second constraint is needed here for preventing structures as in Ex. (2.27), but also follows from the independently motivated principle of *economy of expression* (Bresnan, 1995, 2001), which constrains constituency trees to have as few non-terminal nodes as the grammar allows.

(2.26) X → Y X*

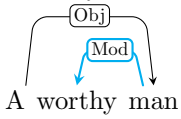
XP → XP* YP

(2.27) * A man worthy.

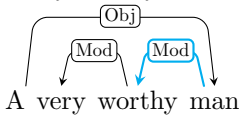


Dependency grammars are unable to distinguish between the word level and phrase level to the same end as phrase structure grammars. While one might be able to formalize a constraint along the lines of "modifiers with no dependents go before the word they modify, but modifiers with dependents go after it", this fails to capture the observation that intensified adjectives—which *do* have a modifier—must still precede their head.

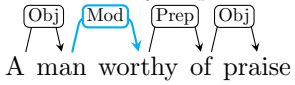
(2.28) A worthy man.



(2.29) A very worthy man.



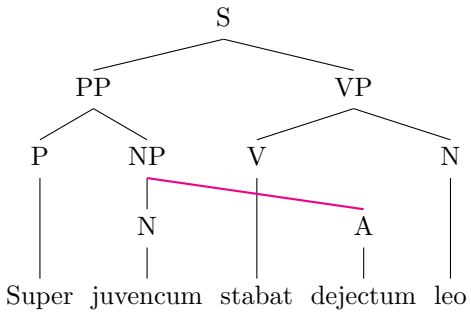
(2.30) A man worthy of praise.



It's not clear what the best way to account for this phenomenon is in a dependency grammar. Phrase structure grammars allow for an elegant analysis of this class of word order constraints that dependency grammars struggle to capture.

While phrase structure grammars are good at explaining rigid word order, dependency grammars are better equipped for explaining relaxed word order. In many languages, word order is so free that grouping words into contiguous phrases with well-defined behavior is difficult or impossible. Latin is one well-known example of this (Ex. 2.31). Languages such as Latin are known to base word order on information structure rather than constituency constraints.

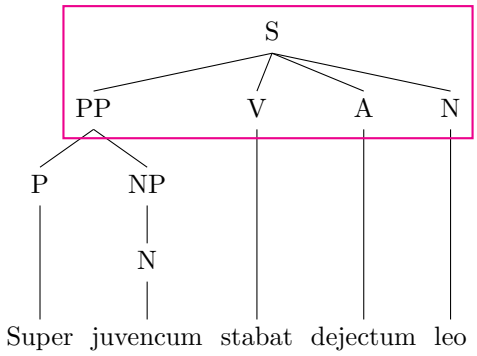
(2.31) Super juvencum stabat dejectum leo
on bull.ACC stood fallen.ACC lion.NOM
'On a fallen bull stood a lion.'



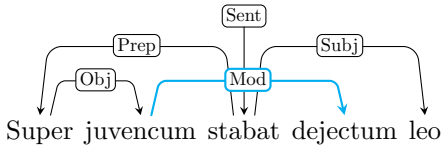
Here, the noun phrase *juvencum dejectum*, "fallen bull" is separated by the verb *stabat*, "stood". However, the main verb of the sentence does not form a logical unit with the object of the preposition phrase. Therefore, the noun phrase is discontinuous, and cannot be modeled with a context-free phrase structure tree. Phrase structure grammars will typically either employ movement or movement-like mechanisms to effectively re-order a more convenient underlying word order into the observed one, or else abandon phrase structure altogether and posit a constituent without immediate internal hierarchy (Ex. 2.32a). By contrast, dependency grammars don't generally make the assumption of surface-level contiguity. This means that in a dependency analysis of Ex. (2.32b), the offending adjective *dejectum* can depend directly on the noun it modifies without complication, which is impossible in a phrase structure tree.

(2.32) Super juvencum stabat dejectum leo.

a.



b.



While dependency grammars allow crossing branches, it's fact that most dependencies in most languages involve non-crossing branches. So just as how phrase structure grammars have to explain

why most phrases have heads and dependency grammars have to explain why some phrases don't, here dependency grammars have to explain why most phrases don't cross dependencies and phrase structure grammars have to explain why some phrases do.

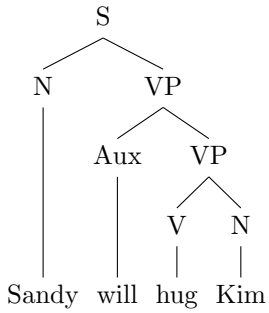
One observation that follows from this discussion is that phrase structures are very effective for describing why certain word orders aren't allowed, whereas dependency grammars are very effective for describing why other word orders are. In other words, phrase structure frameworks tend to be more restrictive, whereas dependency structure frameworks tend to be more permissive. The difference is consequential when trying to design a general-purpose, computationally practical syntactic representation. In order to be general-purpose, it would need to have sensible analyses for the kind of awkward, slightly ungrammatical sentences common in spoken language and web text. A formalism that emphasizes restrictions on allowable constructions will fail to find reasonable structures for these sentences more often than one that emphasizes allowances. This suggests that even if phrase structures are better at characterizing surface-level linguistic structures theoretically, their exclusivity may hinder their usefulness as a computational tool.

2.2 From Transformational Grammar to the Penn Treebank

The Penn Treebank (Marcus et al., 1993) was designed to provide a large corpus of linguistically annotated written natural language, and has been the most widely used resource for computational research on English syntax. The goal of this section is to give the reader an overview of the theory behind the Penn Treebank constituency structure representation—both the official one and the one produced by most statistical parsers. Early Transformational Grammar (TG; Chomsky 1965) has heavily influenced most modern syntactic theories. This framework aims to characterize a set of rules that generates all and only the acceptable sentences in a language. The generation procedure begins with a context-free phrase structure grammar that produces the *deep structure* of a sentence, which specifies its semantic content, and then uses context-sensitive rewrite rules to convert the deep structure into the sentence's *surface structure*, the string of words that ultimately gets spoken (or written).

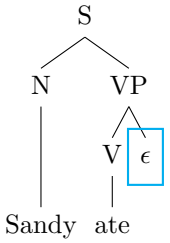
Ideally, all sentences with the same semantic content should share the same deep structure. To this end, semantic predicates have the structure in Ex. (2.33).

(2.33) Sandy will hug Kim.

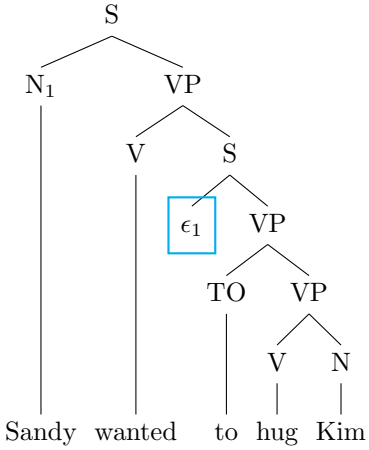


The whole predicate is under an S node, with the agent highest, optionally followed hierarchically by any auxiliaries, then the verb itself at the bottom, leftwards of any non-agent arguments or optional adjuncts. Sometimes, though, a predicate will be missing an argument. These missing arguments can be represented with an empty token, notated here ϵ (Ex. 2.34). If the argument appears elsewhere in the sentence, as in most control constructions, it can be linked back to the predicate it's missing from by indexing ϵ and the overt appearance (Ex. 2.35).

(2.34) Sandy ate.



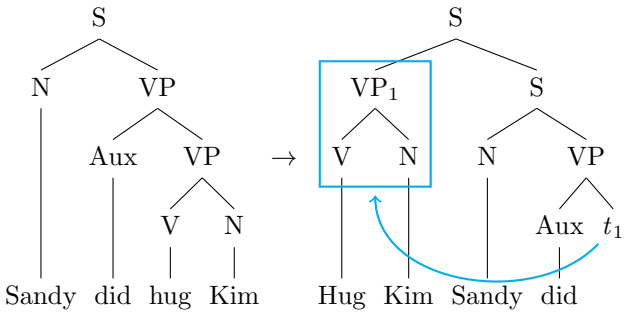
(2.35) Sandy wanted to hug Kim.



Converting from deep structure to surface structure involves context-sensitive rewrite rules. VP topicalization involves this kind of rewriting. First the un-topicalized sentence is generated, then the verb phrase is moved to the front of the sentence. It is often convenient to mark where the

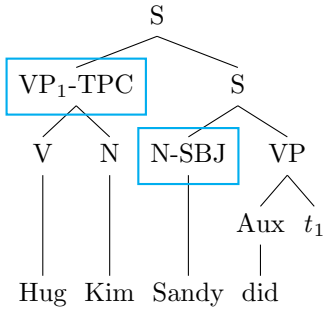
dislocated argument originated from; this is annotated with a silent *trace* (which is distinct from ϵ) co-indexed with the phrase that moved.

(2.36) Hug Kim Sandy did.



These constituency trees, with traces `*t*` for moved constituents and empty strings `*` for missing ones, are the basis of the representation used by the Penn Treebank (Marcus et al., 1993). Because the trees include empty tokens and coindexed traces, the deep structure of a sentence is fully recoverable. However, phrase structure nodes mark only the syntactic category of the head of the constituent and whether the constituent is a leaf node. Consequently, the constituency tree can be difficult to read, and the way that the meaning of a sentence is constructed from the structure can be opaque. In order to turn the constituency representation into something more interpretable and useful for downstream applications, subsequent releases of the Penn Treebank also annotate some categories with the grammatical function that the constituent plays in the sentence. Subjects, topicalized phrases, locative phrases, temporal phrases, nonverbal predicates, and nominalized verbs are all annotated in the phrase structure tree. An example is shown in Ex. (2.37).

(2.37) Hug Kim Sandy did.

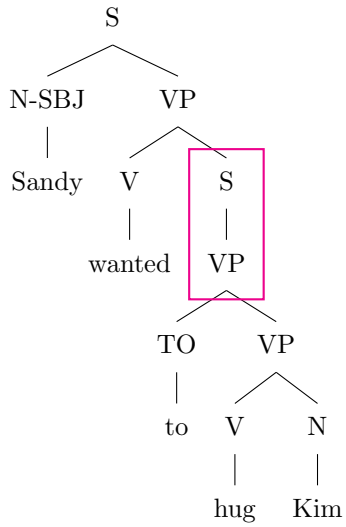


Here the topicalized verb phrase is explicitly marked as the topic, and the subject of *hug* is similarly marked as a subject.

The empty tokens ϵ and t are important for reconstructing the meaning of a sentence and linking dislocated arguments to predicates; however, they are unnecessary for statistical parsing, serving only to complicate the tree. In part because of the difficulty associated with reproducing them, and in part by historical convention, many parsers remove them at training time and make no attempt

to generate them. While the resulting trees are easier to create, the relationship between deep structure and surface structure is somewhat obfuscated. Infinitival constructions are perhaps the best example of this. When the S category is interpreted as representing a whole semantic predicate at deep structure, the structure of sentences like Ex. (2.35) is clearly justifiable. However, when the empty nodes are removed, as in Ex. (2.38), it's more natural to intrepret the S node as representing an inflected sentence. Control constructions bear a predicate, meaning they appear under an S in the tree, but they don't syntacically behave like inflected clauses. Consequently, the S node in infinitival constructions is at best redundant and at worst confusing.

(2.38) Sandy wanted to hug Kim.



Many parsers likewise omit the functional annotations. The original version of the Penn Treebank had already become the standard when they were added in the second version, so historical convention and inertia hindered their use. Additionally, including functionally annotated categories as distinct from their unannotated counterparts sparsifies the training data. The number of possible syntactic rules increases significantly, and furthermore the number of attested instances of each one decreases. For early statistical parsers, this caused accuracy to suffer too much for them to be adopted generally, although some work has found aspects of them to be useful (Klein and Manning, 2003).

The point of this discussion is that the Penn Treebank is a theoretically motivated representation, deriving from research on English syntax in the early days of Transformational Grammar. While TG has changed substantially in the decades since the inception of the Penn Treebank, the structures don't deviate fundamentally from what the current theory would dictate. However, some aspects of the Penn Treebank representation have been simplified to make it easier for users to interpret or because machine learning researchers deemed them unnecessary (e.g. traces), or to make it easier to train accurate statistical parsers on (e.g. functional tags). This points to the utility of having a simple

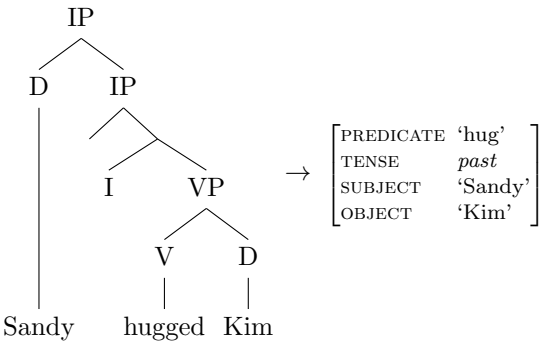
representation that can be easily understood by researchers without a strong linguistic background and parsed relatively easily. The following sections will consider alternative representations derived from linguistic theory that have been argued to be simpler to understand, parse, and utilize.

2.3 From Lexical Functional Grammar to Universal Dependencies

This section is intended to provide the reader with a deeper understanding of the theoretical origins of the Universal Dependencies framework. Lexical Functional Grammar (LFG; Kaplan and Bresnan 1982) developed as a response to some of the assumptions made by Transformational Grammar. TG can be summarized as representing semantic information with a constituency tree, then using rewrite rules to convert the deep constituency tree into a surface constituency tree. Lexical Functional Grammar, on the other hand, finds that constituency trees are an inefficient and unenlightening way to represent abstract information. The fact that deep structure is ambiguous and can yield many different surface structures is likewise worrying, since it means that deep structure is insufficient for representing all the information contained in a sentence (such as topic and focus). Furthermore, LFG takes issue with the use of rewrite rules on empirical, computational, and psychological bases. Instead, it uses a separate data structure altogether—an unordered dictionary known as an *attribute-value matrix*—to represent the deep structure of a sentence. This object contains functional rather than semantic information, including primitive attributes such as *predicate*, *subject*, *object*, *topic*, and *modifiers*. Because LFG's interpretation of the latent structure in a sentence—known in the theory as *functional structure*—includes information structure, different sentences will have different functional representations. However, *similar* sentences will still have *similar* functional structures, even if their constituent structures differ dramatically. The functional structure of a sentence is constructed directly from the constituency-level context-free grammar, which includes rules dictating how the constituents of a phrase combine together. Additional rules and constraints can then map the functional parse to a semantic parse to determine the semantic content of a sentence.

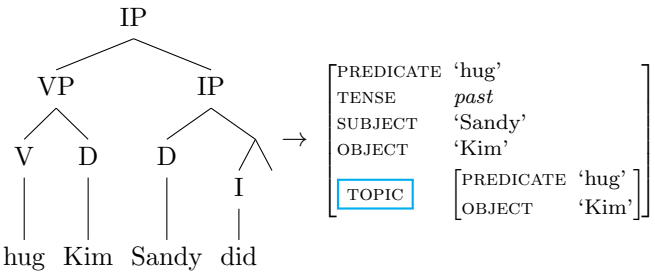
As with Transformational Grammar, LFG aims to characterize the class of sentences that are and aren't allowed in a given language. Unlike Transformational Grammar, which uses a CFG to characterize deep structure and accepts any surface structure that can be transformed from it, LFG uses a CFG to characterize *surface structure* and accepts any sentence that can be assigned a valid functional parse. The nature of the functional representation is normally taken to be the primary focus of interest. In this section, the consituency trees will more or less follow conventions adopted in the LFG literature, with the inflection phrase (IP) representing an inflected sentence (S), with the intermediate "bar" category omitted, and with names being the same category as determiners (D).

(2.39) Sandy hugged Kim.

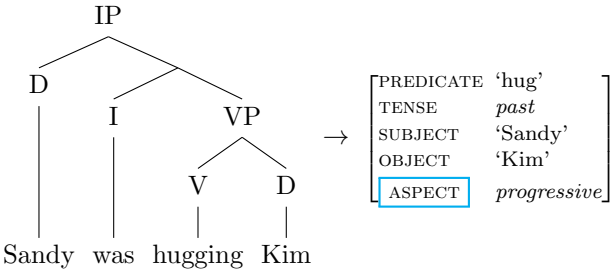


In Ex. (2.39), the main verb, subject, object, and tense are all identified in the constituency structure and extracted into the functional representation. Critically, this functional structure is almost identical to variants that involve a substantially manipulated constituency structure, as shown in Ex. (2.40, 2.41).

(2.40) Sandy wanted to hug Kim, and hug Kim Sandy did.

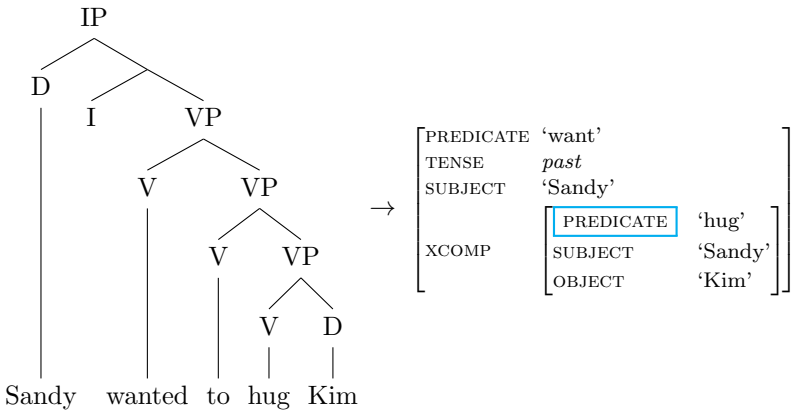


(2.41) Sandy was hugging Kim.



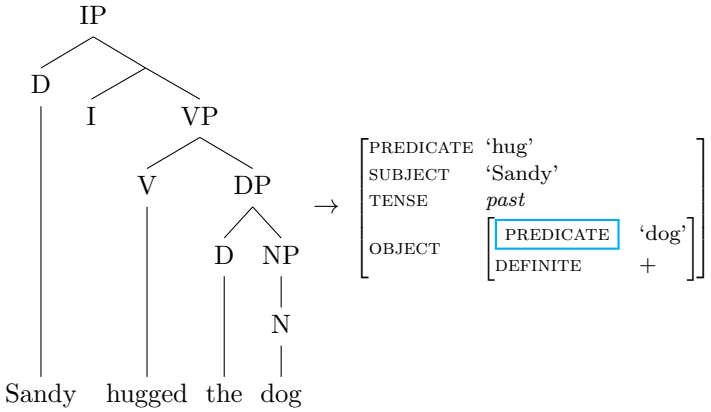
The main predicate, subject, object, and tense of the functional structure is consistent across Exs. (2.39–2.41), even though the constituency structure is different. This additional transparency makes the functional structure an appealing alternative to constituency structure as a representation of the abstract content of a sentence. Note that as with constituency trees, this functional representation allows for predicates with their modifiers to be nested hierarchically. Nested predicates always have their own attribute-value matrix that contains the grammatical functions that pertain to them. In Ex. (2.42), for example, the word *hug* is a predicate that modifies *want*, so it gets assigned its own attribute-value matrix with its own arguments and adjuncts.

(2.42) Sandy wanted to hug Kim.



One final relevant feature is that not only verbs can introduce predicates. Common nouns, adjectives, and adverbs are all also normally assumed to have predicates. The common noun *dog* is shown in Ex. (2.43) to introduce a new predicate in spite of not being verbal in nature.

(2.43) Sandy hugged the dog.



While this representation is closer to semantics in a number of ways, it isn't intended to be a semantic representation. The subject of an active sentence and the subject of a passive sentence will both be assigned to the SUBJECT function of the predicate, even though they generally have different semantic roles. This is by design, to avoid removing important non-semantic information from the representation, such as information structure or social meaning. It also means that the functional structure doesn't need to represent non-functional semantic information, such as quantifier scope (see Section 2.4 for further discussion).

The invariance of LFG's functional structure to changes in phrase structure that have little impact on meaning makes it appealing as the basis for an alternative to constituency structure in computational tasks. For this reason, the *Stanford Dependencies* (SD; De Marneffe et al. 2006)

syntactic annotation system draws heavily from the ideas of functional structure. In essence, Stanford Dependencies aims to annotate the functional structure of a sentence in a way that's relatively easy for people without linguistic training to understand as well as straightforward to parse and use in NLP systems. In order to do this, it takes two generalizations about functional structure and enforces them as hard constraints. The first is that each word in the sentence normally modifies exactly one predicate. The second is that each word in the sentence contributes more or less one grammatical function to that predicate. Thus SD forces each word to contribute exactly one function to exactly one other word's predicate. This allows functional structure to be represented as direct, *bi-lexical* relationships between word pairs, with each word implicitly taking the place of the content it provides to functional structure. Ex. (2.44) provides a concrete example.

(2.44) Sandy wanted to hug Kim.

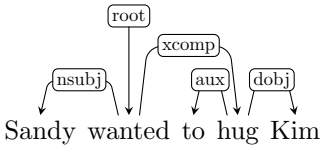
$$\begin{bmatrix} \textsc{predicate} & \text{'want'} \\ \textsc{tense} & \textit{past} \\ \textsc{subject} & \text{'Sandy'} \\ \textsc{xcomp} & \begin{bmatrix} \textsc{predicate} & \text{'hug'} \\ \textsc{subject} & \text{'Sandy'} \\ \textsc{object} & \text{'Kim'} \end{bmatrix} \end{bmatrix} \rightarrow \begin{bmatrix} \textsc{nominal subject} & \text{'Sandy'} \\ \textsc{predicate} & \text{'wanted'} \\ \textsc{xcomp} & \begin{bmatrix} \textsc{auxiliary} & \text{'to'} \\ \textsc{predicate} & \text{'hug'} \\ \textsc{direct object} & \text{'Kim'} \end{bmatrix} \end{bmatrix}$$

In Ex. (2.44), the functional representation as formulated by LFG is tweaked and simplified to generate the Stanford Dependencies version. Information about tense is no longer explicitly represented, instead being left inferrable from auxiliaries or the morphological form of the predicate. Subjects are marked for their categorial status, being either nominal (NSUBJ) or clausal (CSUBJ). Nominal objects are classified as being either direct (DOBJ) or indirect (IOBJ). Some functionally vacuous words such as the auxiliary *to* are included only in the SD representation. *Sandy* is only explicitly marked as the subject of *wanted*, not as the subject of *hug*. On the whole, however, these are very small differences; the two structures contain most of the same information organized in a similar manner.

The attribute-value matrix representation is convenient for linguistic theory but are difficult to read—especially for long sentences—and impractical for downstream systems. Because each word explicitly modifies (or *depends on*) exactly one predicate, and each predicate is introduced by exactly one word, there is a one-to-one mapping of words in the sentence to grammatical functions in the structure. This means SD's functional structure can be visualized with a dependency tree (Ex. 2.45).

(2.45) Sandy wanted to hug Kim.

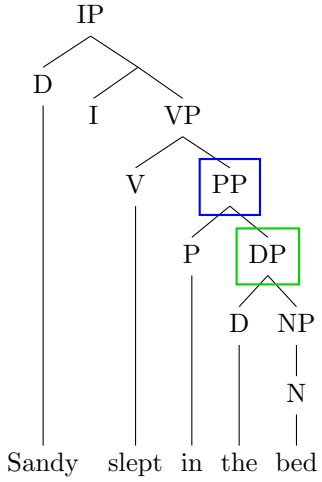
$$\begin{bmatrix} \textsc{predicate} & \text{'wanted'} \\ \textsc{nominal subject} & \text{'Sandy'} \\ \textsc{xcomp} & \begin{bmatrix} \textsc{predicate} & \text{'hug'} \\ \textsc{auxiliary} & \text{'to'} \\ \textsc{direct object} & \text{'Kim'} \end{bmatrix} \end{bmatrix} \rightarrow$$



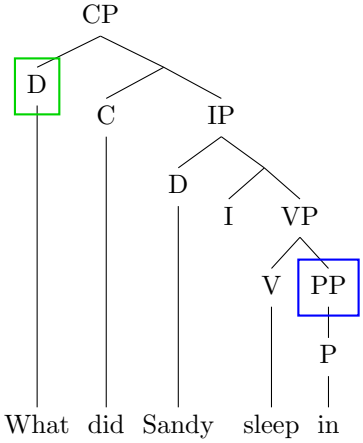
The highest-level predicate is labeled as the *root* of the structure, and all other predicates are labeled according to the immediately higher grammatical function (e.g. *hug* is labeled *xcomp* in Ex. (2.45)), but otherwise the two visualizations are isomorphic.

The Stanford Dependencies annotation scheme explicitly annotates all grammatical functions, rather than leaving them inferrable from constituency structure, and removes superfluous hierarchy relevant only to constituency structure. It should be apparent that this makes it generally more transparent than the Penn Treebank representation. However, it still has a few drawbacks. One of the most glaring limitations is that it was designed specifically for English, and some of its analyses are suboptimal for other languages. The clearest example of this relates to prepositional phrases. In English, some common predicate-like relationships are marked syntactically using prepositions like *in* or *of*. There is strong evidence that at the level of constituency structure, these words join with a complete noun phrase to create a prepositional phrase PP—for instance, in English the noun or noun phrase can be dislocated, leaving the preposition behind.

(2.46) Sandy slept in the bed.

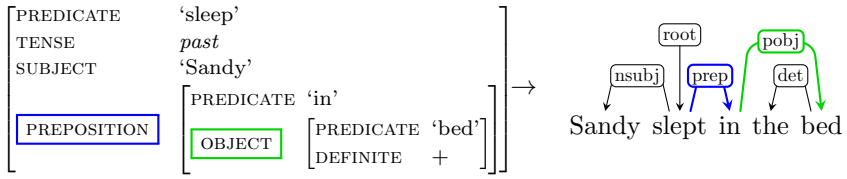


(2.47) What did Sandy sleep in?



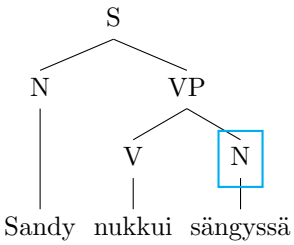
Following the traditional LFG functional representation, SD take the noun to be the dependent of the preposition.

(2.48) Sandy slept in the bed.



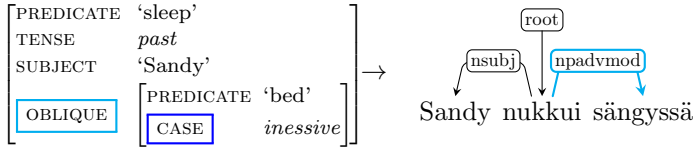
Not all languages mark these relationships syntactically—many instead mark them *morphologically*, using case suffixes. Finnish is one language that does so.

(2.49) Sandy nukku-i sängy-ssä
Sandy sleep-PAST bed-LOC
'Sandy slept in the bed'



The English SD annotation scheme is unequipped to handle this phenomenon. On the one hand, the word contains the same functional content as the whole prepositional phrase in English, suggesting that it should be assigned the *prep* label. On the other hand, the word is a noun modifying a verb with no other word to depend on, suggesting that it should be assigned the *npadvmod* (noun phrase adverbial modifier) label. The standard LFG analysis is closest to the latter.

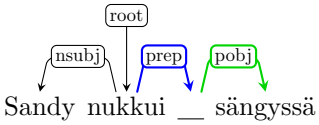
(2.50) Sandy nukkui sängyssä.



In either case, the oblique modifier depends directly on the verb in Finnish, but is separated by an intermediate dependent in English. This means that two sentences with nearly identical functional content would have very different functional representations, counter to the principles of LFG and SD.

Modifying Finnish SD to make the paths the same as English SD would require inserting an empty token in the parse whose only purpose is to add an edge to the tree. While this solves the problem of having different representations for what is essentially the same relationship in two languages, it violates SD's goal of having a one-to-one mapping of words to grammatical functions. It is also unappealing linguistically, since the idea that there are an abundance of unspoken prepositions in the language would be unnatural to most native Finnish speakers. This would likewise make SD difficult for statistical parsers to work with, for the same reasons empty tokens are generally abandoned in parsers trained on the Penn Treebank.

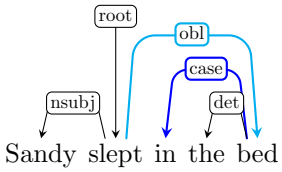
(2.51)



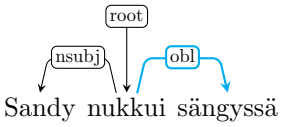
The alternative solution is to modify English SD rather than the hypothetical Finnish SD. In this approach, the noun would be made to depend on the main verb in both languages, and in English

the preposition would depend on the noun.

(2.52) a.



b.



This solution has the additional advantage of decreasing the path length between content words, so that verbs and nouns that modify them are no longer separated by an extra dependent. Furthermore, the LFG functional structure analogous to the English SD representation is perfectly valid and defensible (albeit somewhat nonstandard), and retains the appealing property of being structurally similar to the Finnish version.

(2.53) a. Sandy slept in the bed.

$$\begin{bmatrix} \text{PREDICATE} & \text{'sleep'} \\ \text{TENSE} & \textit{past} \\ \text{SUBJECT} & \text{'Sandy'} \\ \boxed{\text{OBLIQUE}} & \begin{bmatrix} \text{PREDICATE} & \text{'bed'} \\ \text{DEFINITE} & + \\ \boxed{\text{ADPOSITION}} & [\text{PREDICATE } \text{'in'}] \end{bmatrix} \end{bmatrix}$$

b. Sandy nukkui sängyssä

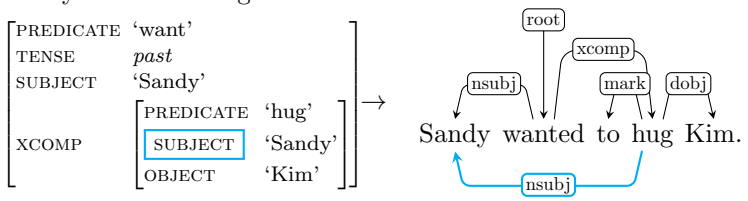
$$\begin{bmatrix} \text{PREDICATE} & \text{'sleep'} \\ \text{TENSE} & \textit{past} \\ \text{SUBJECT} & \text{'Sandy'} \\ \boxed{\text{OBLIQUE}} & \begin{bmatrix} \text{PREDICATE} & \text{'bed'} \\ \boxed{\text{CASE}} & \textit{inessive} \end{bmatrix} \end{bmatrix}$$

The utility of having an annotation scheme that can be sensibly applied to any language is what motivated the development of *Universal Dependencies* (UD; Nivre et al. 2016). Universal Dependencies tweaks and extends Stanford Dependencies to be useful for languages with substantially different properties and orthographic conventions from English. Not only is the goal of having a cross-linguistic representation desirable from both a researcher and user perspective, but it also aligns very well with the original goals of LFG, which developed in part as a reaction to the largely English-centric focus of theoretical syntax at the time of its inception. In addition to changing the analyses for a few phenomena to make them more cross-linguistically natural, UD tries to provide other cross-linguistic annotations. Different languages can have radically different part-of-speech tag conventions, for instance, so a complete UD annotation of a sentence will include both the language-specific POS tag annotations as well as coarse-grained language-independent POS tags. UD also annotates inflectional morphological information (such as tense and definiteness) for both theoretical and practical reasons. In LFG, these morphological features are present in the functional structure, but for simplicity were removed from the SD representation. By annotating words for morphology, UD effectively brings these features back into the framework, making the implementation closer to the original theory. Practically speaking, these morphological features can be useful for downstream tasks, either as features or embeddings to be weighted in a machine learning system or to help guide rule-based information extraction. UD does currently lack information structure in its representation for most languages, largely because of how difficult it is to annotate. As of Universal Dependencies version 2.2 (Nivre et al., 2018), UD has been applied to 71 different languages.

Another limitation of the version of SD/UD described thus far is that it requires the dependency

annotation to be a tree, with every word depending on exactly one other word. However, some syntactic constructions involve one word modifying two predicates. In the control sentence *Sandy wanted to hug Kim*, *Sandy* intuitively depends on two words—*wanted* and *hug*—but SD only permits one of these two relationships to be represented. In order to include both at once, the tree requirement needs to be relaxed. SD and UD address this by including an *enhanced* representation (also known as the *collapsed* representation in SD), which includes relationships that can't be present in the tree-based *basic* representation. Historically, this non-strictly-tree-structured formalism was proposed first, and later simplified into the strictly tree-structured version (De Marneffe and Manning, 2008)

(2.54) Sandy wanted to hug Kim.



This allows Universal Dependencies to capture even more information present in an LFG functional structure, even when one word contributes grammatical functions to multiple predicates. The following sections discuss other dependency formalisms that likewise relax the assumption of tree structure. While UD only claims to be syntactic in nature, these representations aim to use dependency structures to annotate the semantic relationships in a sentence.

2.4 From Minimal Recursion Semantics to DELPH-IN MRS

Stanford Dependencies and Universal Dependencies are derived from Lexical Functional Grammar, but there are other formal theories of linguistics that include representations that can be cast into a dependency framework. Head-driven Phrase Structure Grammar (HPSG; Pollard and Sag 1994) and its successor Sign-Based Construction Grammar (SBCG; Sag 2012) are syntactic theories that parse sentences into a semantic representation that has been used as the basis for its own dependency formalism. As with TG and LFG, these frameworks are designed to characterize the acceptable sentences of a language. Their approach is most similar to LFG's—they accept any sentence that can be assigned a valid parse, and they include mechanisms for generating a semantic denotation of a valid sentence. They differ from LFG in that they represent the whole parse, including semantic and constituency information, in a single hierarchical attribute-value matrix. The theory of semantics that HPSG and SBCG espouse—and which will be detailed here—is known as *Minimal Recursion Semantics*. Minimal Recursion Semantics (MRS; Copestake et al. 2005), an extension of Hole Semantics (Bos, 1996), serves as the inspiration for *DELPH-IN Minimal Recursion Semantics*, one of three semantic dependency formalisms introduced in the SemEval shared task on semantic

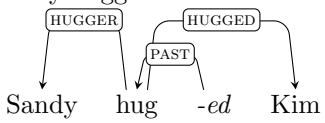
dependency parsing (SemEval; Oepen et al. 2014). In this representation of meaning, the content of a sentence is made up of a set of semantic predicates (or *frames*) that are linked together by indices. For example, in the sentence *Sandy hugged Kim* (Ex. 2.55), two entities named *Sandy* and *Kim* are introduced, and indexed with the variables i and j. The statement also introduces a *hugging* situation, indexed with the event variable s, where *Kim* is the HUGGER and *Sandy* is the HUGGED person. The morphology of the verb indicates that the event took place in the past, so there is also a *past* frame that modifies the *hugging* event.

(2.55) Sandy hugged Kim.

$$\left\{ \begin{bmatrix} \text{RELATION} & \textit{name} \\ \text{ENTITY} & i \\ \text{NAME} & \text{`Sandy'} \end{bmatrix}, \begin{bmatrix} \text{RELATION} & \textit{name} \\ \text{ENTITY} & j \\ \text{NAME} & \text{`Kim'} \end{bmatrix}, \begin{bmatrix} \text{RELATION} & \textit{hug} \\ \text{SITUATION} & s \\ \text{HUGGER} & i \\ \text{HUGGED} & j \end{bmatrix}, \begin{bmatrix} \text{RELATION} & \textit{past} \\ \text{PAST} & s \end{bmatrix} \right\}$$

As with LFG's functional structure, the semantic predicates can be organized into a dependency structure. The DELPH-IN Minimal Recursion Semantics (DM; Flickinger et al. 2012; Oepen et al. 2014) representation makes this its goal. In Ex. (2.56), each word stands in for the predicate it introduces, with the past tense morpheme *-ed* representing the *past* frame (for now). Every word that introduces an ENTITY or SITUATION index then depends on every word that references it. *Sandy* introduces the ENTITY variable i, and *hug* references i in its HUGGING relation, so *Sandy* is made to depend on *hug* and labeled the HUGGER.

(2.56) Sandy hugged Kim.

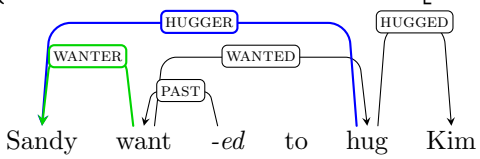


This hierarchical visualization of the semantic structure contains almost all the same information as the "flat" list approach. Only the RELATION attribute of each frame is lost, but this is generally recoverable from the individual lexical items.

Variables can be referenced by multiple frames, so by the above definition, a word can have multiple head words that it depends on. This is in contrast to the PTB, SD, and basic UD frameworks, where every word has one and only one head. Ex. (2.57) provides an example.

(2.57) Sandy wanted to hug Kim.

$$\left\{ \begin{bmatrix} \text{RELATION} & \textit{name} \\ \text{ENTITY} & i \\ \text{NAME} & \text{`Sandy'} \end{bmatrix}, \begin{bmatrix} \text{RELATION} & \textit{name} \\ \text{ENTITY} & j \\ \text{NAME} & \text{`Kim'} \end{bmatrix}, \begin{bmatrix} \text{RELATION} & \textit{hug} \\ \text{SITUATION} & s_1 \\ \boxed{\text{HUGGER}} & i \\ \text{HUGGED} & j \end{bmatrix}, \begin{bmatrix} \text{RELATION} & \textit{want} \\ \text{SITUATION} & s_2 \\ \boxed{\text{WANTER}} & i \\ \text{WANTED} & s_1 \end{bmatrix}, \begin{bmatrix} \text{RELATION} & \textit{past} \\ \text{PAST} & s_2 \end{bmatrix} \right\}$$



Here, *Sandy* (again indexed by i) is marked as being both the WANTER and HUGGER, and as such depends on both *want* and *hug*.

One problem that arises when developing a framework to formalize the meaning of any given sentence in a language is that many sentences can have multiple meanings depending on the context. For example, Ex. (2.58) can have two meanings depending on the clause that follows it.

(2.58) Every student already knows an answer, because ...

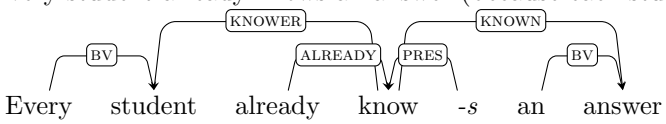
a. each student wrote a question for the quiz.

b. it was accidentally leaked by the TA.

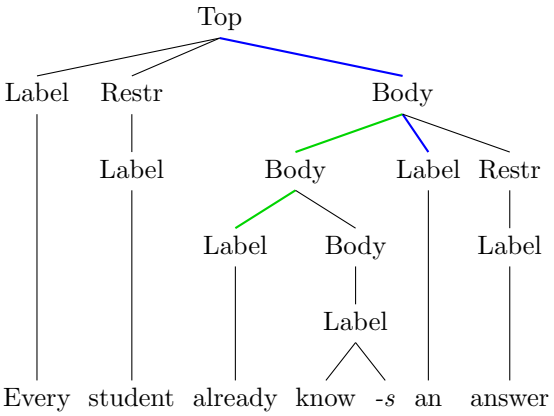
In Ex. (2.58a), each student knows a different answer (the one they wrote), but in Ex. (2.58b), each student knows the same answer (the one that was leaked). The two readings of Ex. (2.58) arise from the two ways that the universal quantifier *every* and the existential quantifier *a* can interact. This is known as *scopal ambiguity*. Scopal ambiguity raises two issues: how should the framework represent the two fully resolved interpretations, and how should it represent the ambiguity of the sentence before it gets resolved?

Traditionally, scopal interactions have been modeled by organizing scope-bearing elements (or *operators*) hierarchically in a tree structure. Entities or events instantiated by lower scopal operators are created for each entity or event that was instantiated by a higher one. In Ex. (2.58a)—where the universal quantifier *every* has higher scope than the existential *a*—first n students are instantiated, then for each student i, exactly one answer that student i knows is instantiated. In Ex. (2.58b)—where the hierarchy is reversed—first one answer j is instantiated, followed by n students who know that answer j. In the minimal recursion semantics framework, the two interpretations of the sentence are represented by *labeling* each frame and marking which ones a scope-bearing element immediately outscopes. Note that determiners are not the only scopal operators, as a number of adverbs—including *almost* and *already*—are as well. Quantifiers are generally assumed to scope over two subtrees: the *restriction*, which describes the kind of entity being instantiated; and the *body*, which explains how the entity or entities relate to the rest of the meaning. They are also assumed to "bind" to a variable, either an entity or a situation, which connects it to the dependency graph of entities and situations. Finally, predicates that semantically modify other predicates but don't introduce any scope of their own are assigned the same label as the frame they modify. All this ensures that every frame is part of the scopal structure. For largely practical purposes, it is useful to mark the frame at the top of the scopal hierarchy. The two readings are shown with a flat, frame-based representation and a hierarchical tree-based representation in Ex. (2.59, 2.60).

(2.59) Every student already knows an answer (because each student wrote one).

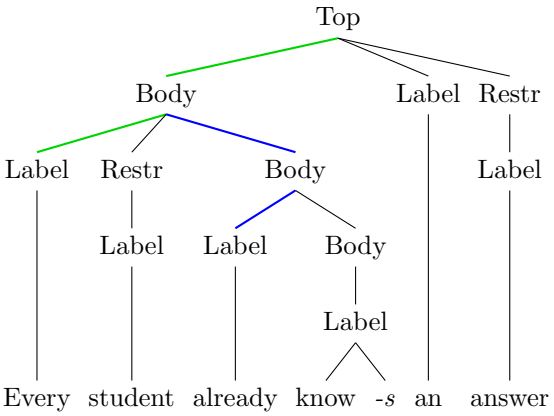


$$\begin{bmatrix} \text{TOP} & l_1 \\ \text{FRAMES} & \left\{ \begin{bmatrix} \text{RELATION} & every \\ \text{LABEL} & l_1 \\ \text{BV} & i \\ \text{RESTR} & l_2 \\ \text{BODY} & \boxed{l_3} \end{bmatrix}, \begin{bmatrix} \text{RELATION} & student \\ \text{LABEL} & l_2 \\ \text{ENTITY} & i \end{bmatrix}, \begin{bmatrix} \text{RELATION} & a \\ \text{LABEL} & \boxed{l_3} \\ \text{BV} & j \\ \text{RESTR} & l_4 \\ \text{BODY} & \boxed{l_5} \end{bmatrix}, \begin{bmatrix} \text{RELATION} & answer \\ \text{LABEL} & l_4 \\ \text{ENTITY} & j \end{bmatrix}, \right. \\ & \left. \begin{bmatrix} \text{RELATION} & already \\ \text{LABEL} & \boxed{l_5} \\ \text{ALREADY} & s \\ \text{BODY} & l_6 \end{bmatrix}, \begin{bmatrix} \text{RELATION} & know \\ \text{LABEL} & l_6 \\ \text{SITUATION} & s \\ \text{KNOWER} & i \\ \text{KNOWN} & j \end{bmatrix}, \begin{bmatrix} \text{RELATION} & present \\ \text{LABEL} & l_6 \\ \text{PRESENT} & s \end{bmatrix} \right\} \end{bmatrix}$$



(2.60) Every student already knows an answer (because it was leaked).

$$\begin{bmatrix} \text{TOP} & l_3 \\ \text{FRAMES} & \left\{ \begin{bmatrix} \text{RELATION} & every \\ \text{LABEL} & \boxed{l_1} \\ \text{BV} & i \\ \text{RESTR} & l_2 \\ \text{BODY} & \boxed{l_5} \end{bmatrix}, \begin{bmatrix} \text{RELATION} & student \\ \text{LABEL} & l_2 \\ \text{ENTITY} & i \end{bmatrix}, \begin{bmatrix} \text{RELATION} & a \\ \text{LABEL} & l_3 \\ \text{BV} & j \\ \text{RESTR} & l_4 \\ \text{BODY} & \boxed{l_1} \end{bmatrix}, \begin{bmatrix} \text{RELATION} & answer \\ \text{LABEL} & l_4 \\ \text{ENTITY} & j \end{bmatrix}, \right. \\ & \left. \begin{bmatrix} \text{RELATION} & already \\ \text{LABEL} & \boxed{l_5} \\ \text{ALREADY} & s \\ \text{BODY} & l_1 \end{bmatrix}, \begin{bmatrix} \text{RELATION} & know \\ \text{LABEL} & l_6 \\ \text{SITUATION} & s \\ \text{KNOWER} & i \\ \text{KNOWN} & j \end{bmatrix}, \begin{bmatrix} \text{RELATION} & present \\ \text{LABEL} & l_6 \\ \text{PRESENT} & s \end{bmatrix} \right\} \end{bmatrix}$$



The next question is how to represent scopally ambiguous sentences, like Ex. (2.57), without the subsequent context. MRS's approach is to introduce what are called *holes* into the frames. The holes, often notated with h rather than l, essentially function as dummy indices that don't actually refer to specific labels. When sufficient context is present, the holes can be "filled in" or *plugged* with the labels of the appropriate frames. This is done by introducing *constraints* on the denotations (indicated by the CONSTR attribute) that fix the dummy indices to particular frames. In the case of scopal ambiguity, the TOP will be another hole h_0 to be filled in in the CONSTRAINTS feature. This is shown in Ex. (2.61), where the restrictions and bodies of each scopal frame are left underspecified with an h hole and then constrained to be equal to a particular l label.

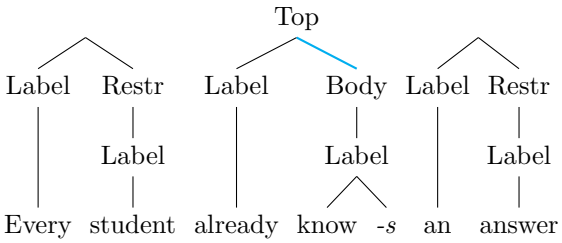
(2.61) Every student already knows an answer (because each student wrote one).

$$\begin{bmatrix}
\text{TOP} & h_0 \\
\text{FRAMES} & \left\{ \begin{bmatrix} \text{RELATION} & every \\ \text{LABEL} & l_1 \\ \text{BV} & i \\ \text{RESTR} & \boxed{h_1} \\ \text{BODY} & \boxed{h_2} \end{bmatrix}, \begin{bmatrix} \text{RELATION} & student \\ \text{LABEL} & l_2 \\ \text{ENTITY} & i \end{bmatrix}, \begin{bmatrix} \text{RELATION} & a \\ \text{LABEL} & l_3 \\ \text{BV} & j \\ \text{RESTR} & \boxed{h_3} \\ \text{BODY} & \boxed{h_4} \end{bmatrix}, \begin{bmatrix} \text{RELATION} & answer \\ \text{LABEL} & l_4 \\ \text{ENTITY} & j \end{bmatrix}, \right. \\
 & \left. \begin{bmatrix} \text{RELATION} & already \\ \text{LABEL} & l_5 \\ \text{ALREADY} & s \\ \text{BODY} & \boxed{h_5} \end{bmatrix}, \begin{bmatrix} \text{RELATION} & know \\ \text{LABEL} & l_6 \\ \text{SITUATION} & s \\ \text{KNOWER} & i \\ \text{KNOWN} & j \end{bmatrix}, \begin{bmatrix} \text{RELATION} & present \\ \text{LABEL} & l_6 \\ \text{PRESENT} & s \end{bmatrix} \right\} \\
\text{CONSTR} & \left\{ \boxed{h_0 = l_1,\ h_1 = l_2,\ h_2 = l_3,\ h_3 = l_4,\ h_4 = l_5,\ h_5 = l_6} \right\}
\end{bmatrix}$$

In order to leave the structure in Ex. (2.61) ambiguous with respect to the quantifier scope, some of the holes can be left unresolved, as in Ex. (2.62). In an underspecified tree, the TOP will be the highest resolved scopal operator.

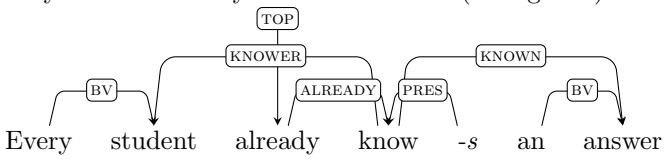
(2.62) Every student already knows an answer (ambiguous)

$$\begin{bmatrix} \text{TOP} & h_0 \\ \text{FRAMES} & \left\{ \begin{bmatrix} \text{RELATION} & every \\ \text{LABEL} & l_1 \\ \text{BV} & i \\ \text{RESTR} & h_1 \\ \text{BODY} & h_2 \end{bmatrix}, \begin{bmatrix} \text{RELATION} & student \\ \text{LABEL} & l_2 \\ \text{ENTITY} & i \end{bmatrix}, \begin{bmatrix} \text{RELATION} & a \\ \text{LABEL} & l_3 \\ \text{BV} & j \\ \text{RESTR} & h_3 \\ \text{BODY} & h_4 \end{bmatrix}, \begin{bmatrix} \text{RELATION} & answer \\ \text{LABEL} & l_4 \\ \text{ENTITY} & j \end{bmatrix}, \right. \\ & \left. \begin{bmatrix} \text{RELATION} & already \\ \text{LABEL} & l_5 \\ \text{ALREADY} & s \\ \text{BODY} & h_5 \end{bmatrix}, \begin{bmatrix} \text{RELATION} & know \\ \text{LABEL} & l_6 \\ \text{SITUATION} & s \\ \text{KNOWER} & i \\ \text{KNOWN} & j \end{bmatrix}, \begin{bmatrix} \text{RELATION} & present \\ \text{LABEL} & l_6 \\ \text{PRESENT} & s \end{bmatrix} \right\} \\ \text{CONSTR} & \{ h_0 = l_5,\ h_5 = l_6 \} \end{bmatrix}$$



Resolving the scope of quantifiers like *a* and *every* actually turns out to be a very difficult task that often requires labor-intensive expert knowledge to annotate and world knowledge to parse. Because of these difficulties, quantifier scope is often excluded from annotation schemes for semantic structures. The DELPH-IN MRS (DM) representation inspired by MRS is no different. In it, the frame at the top of the scopal hierarchy—which may be a predicate or a scopal adverb—is marked with the TOP label, but all other information about scope is lost. Thus the representation in Ex. (2.63) below will be assumed in the following discussion.

(2.63) Every student already knows an answer (ambiguous).

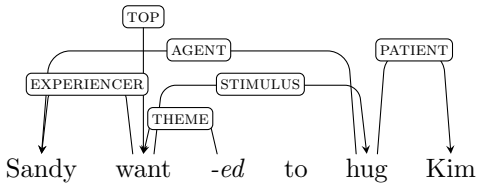


The examples discussed so far have an undesirable property, which is that each frame defines a different set of attributes—WANTER, for example, is unique to *wanting* relations, and HUGGER will only be found in *hugging* ones. There are a few ways to abstract away from these predicate-specific relations. One way is to mark them for a predefined set of *semantic roles*. These roles can include relations like *agent* and *patient*—the entities performing and undergoing the action—as well as *stimulus* and *experiencer*—the entities stimulating and experiencing a psychological response—among a number of others. However, the question of which roles should and shouldn't be included in the set is a matter of ongoing debate, so some approaches to frame semantics involve abstracting even

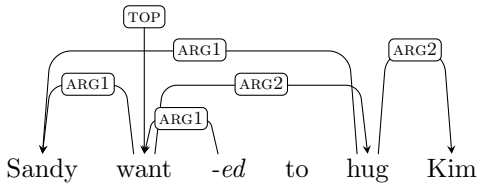
further to create a more theory-neutral system. In MRS, and DM by extension, semantic arguments of frames from predicates are labeled according to a hierarchy of *proto-roles* (cf. propbank labels; Dowty 1991; Palmer et al. 2005), where ARG1 represents agent-like roles, ARG2 represents patient-like roles, and ARG3 represents non-core or beneficiary-like roles. The ENTITY and SITUATION features of the frames—which both effectively serve the same purpose—are replaced with the ARG0 attribute. ARG1 can also be used for adjectival or adverbial frames that modify entities or situations. This allows the formalism to differentiate predicates' arguments without committing to a particular set of semantic roles. The semantic role and proto-role approaches are shown in Ex. (2.64a, 2.64b) respectively.

(2.64) Sandy wanted to hug Kim.

a.

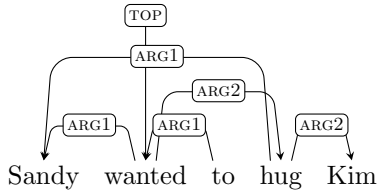


b.



For transparency, the past tense predicate in the list of semantic predicates has also been represented in the dependency structure as the past-tense morpheme up until now. However, to make the dependency parser easier to work with computationally, it's advantageous not to separate roots from their morphemes. Therefore the DM dependency formalism for Minimal Recursion Semantics only relates whole tokens to whole tokens, without segmenting words by morphology.

(2.65)



This restricts the represenational capacity of the formalism in order to make it more practically applicable.

The DM formalism, unlike SD and basic UD, doesn't enforce a tree structure constraint on the dependency structures. Instead, they must be *directed acyclic graphs*. While not entirely unconstrained, this allows the representation to capture more potentially informative bi-lexical relationships in a sentence. Additionally, the relationship between the DM representation of a sentence and that sentence's semantics is arguably more transparently related to semantic theory than in UD.

2.5 From Head-driven Phrase Structure Grammar to Predicate-Argument Structures

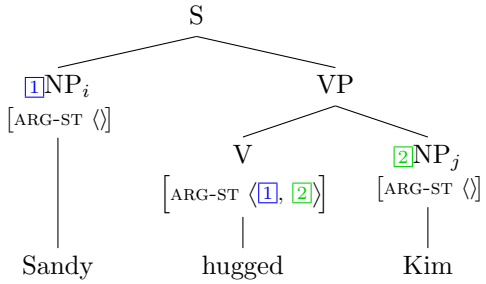
Head-driven Phrase Structure Grammar (HPSG; Pollard and Sag 1994) and Sign-Based Construction Grammar (SBCG; Sag 2012) are designed with the ability to convert a sentence into a coherent semantic structure. They do so by manipulating and constraining words and phrases along three dimensions: constituent structure, semantic structure, and *argument structure*. The more surface-level syntactic representation, constituent structure, contains the phrase structure tree and morphosyntactic features like agreement and morphological form. The semantic structure, as described in the previous section, contains information about meaning and scope, and is the basis of the DM semantic dependency framework. Argument structure, like LFG's functional structure, is an intermediate syntactic representation that imposes constraints on how words and phrases can combine syntactically and facilitates the conversion between constituent syntax and semantics. It is the basis of another semantic dependency formalism, described in this section.

In HPSG, every lexical item ("word" in an abstract sense) has an argument structure list; phrases, on the other hand, do not. For example, the lexical item *hug* has the argument structure in Ex. (2.66a) and the semantic predicate in Ex. (2.66b).

(2.66) a. $\left\langle \text{NP}_i, \text{NP}_j \right\rangle$ b. $\left\langle \begin{bmatrix} \text{ARG0} & s \\ \text{ARG1} & i \\ \text{ARG2} & j \end{bmatrix} \right\rangle$

This indicates that the word's first syntactic argument is the agent-like one performing the action, and the second is the patient-like one receiving the action. It also indicates that both arguments need to be noun phrases. HPSG, like LFG, includes rules and constraints that dictate the relationship between constituent structure and the more abstract representation. In particular, it includes rules that map a verb's arguments to the constituency tree, such that the first argument is in the high *specifier* position to the immediate left of the verb phrase and any following ones are in a flat *complements* list, immediately to the right of the verb.

(2.67) Sandy hugged Kim.



The $\boxed{\#}$ tags mark all structures that need to be identical (or *unifiable* in the presence of underspecificity), which is the primary method HPSG uses to impose constraints on its parses. For compactness, the ARG-ST key will be implicit in the annotations of future trees in this section.

The argument structure of a word can be manipulated to change how its arguments are realized. A transitive verb can be passivized, for instance, by moving the first argument to the end and changing it into an optional prepositional phrase with *by* as the head word. The passive auxiliary *be* in English is then specified as having the same first argument as its second argument, in addition to other constraints on morphology and constituency not relevant here. This ensures that lower verb's subject gets raised to be the subject of the auxiliary. The *argument-marking* preposition *by* has a single NP on its argument-structure list that goes to the right of it in the complement position.

(2.68) hug → hugged

$$\langle \mathrm{NP}_i, \mathrm{NP}_j \rangle \rightarrow \langle \mathrm{NP}_j, (\mathrm{PP}[by]_i) \rangle$$

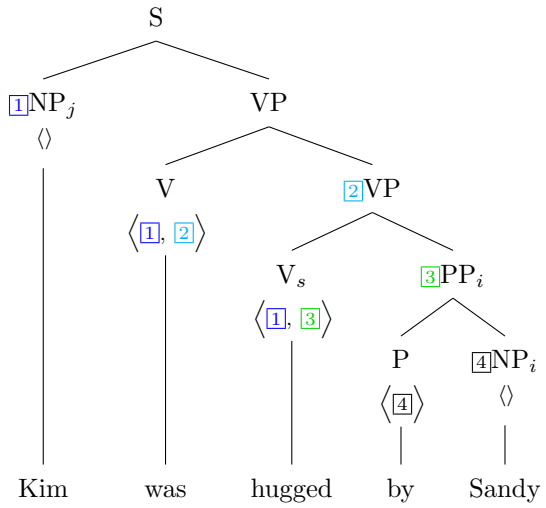
(2.69) passive *be*

$$\left\langle \boxed{1}\mathrm{X}, \begin{bmatrix} \mathrm{VP} \\ \text{ARG-ST} \left\langle \boxed{1} \right\rangle \oplus \boxed{\mathrm{A}} \end{bmatrix} \right\rangle$$

(2.70) argument-marking *by*

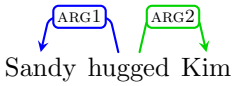
$$\langle \mathrm{NP} \rangle$$

(2.71)

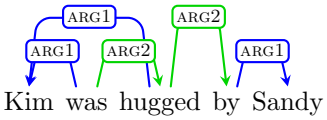


As mentioned above, argument structures are only defined for lexical items, not for phrases. From this it should be easy to see how argument structures can be linked together into a dependency format. Each word depends on all other words that include it (or a phrase that it heads) on their argument structure list, labeled according to its position on that list. Thus ARG1 would go to the first entry on a word's argument structure, ARG2 to its second, etc. This is the basis for the Predicate-Argument Structures (PAS; Miyao and Tsujii 2004; Oepen et al. 2014) dependency formalism.

(2.72)



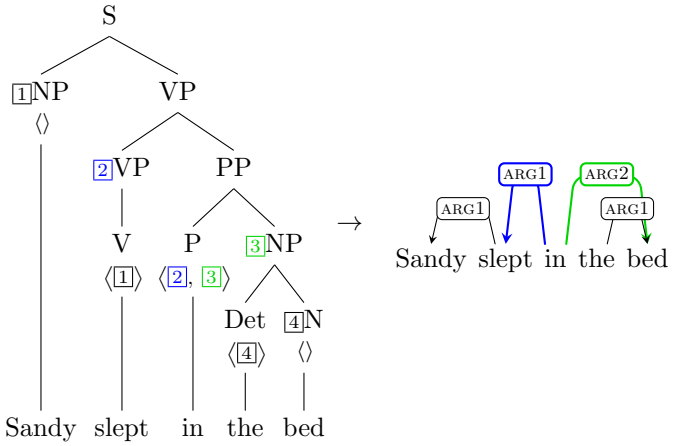
(2.73)



Modifiers, such as adjectives, adverbs, predicative prepositions (but not argument-marking ones),

and determiners, take the word they modify as their first argument at the level of argument structure, and any additional complements as their second. This is normally how modifiers behave in Minimal Recursion Semantics as well.

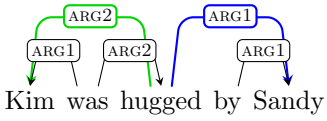
(2.74) Sandy slept in the bed.



As a consequence of making adjuncts heads rather than dependents, the formalisms are radically non-tree-structured. In contrast, enhanced UD is still mostly tree-structured even though it does allow deviations from strict tree-structure.

One problem that arises with the formulation so far is that the agent in an active and passive construction are marked differently. PAS addresses this by marking the dependency structure according to words' *original* argument structure lists, before any lexical rules have been applied. Making this change, the passivized version of Ex. (2.72) would have the structure in Ex. (2.75). Note that the two dependents of *hugged* are now the same as in the active version, with the same labels.

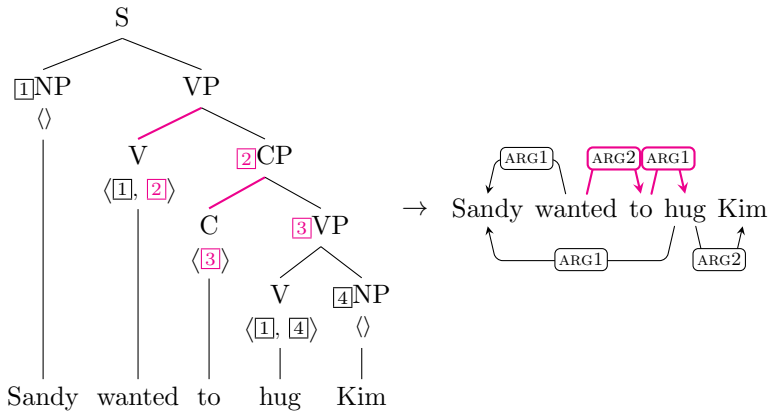
(2.75)



This has the desirable effect of making the representation more abstract, enriching it with information that is harder to recover from the simple string of words.

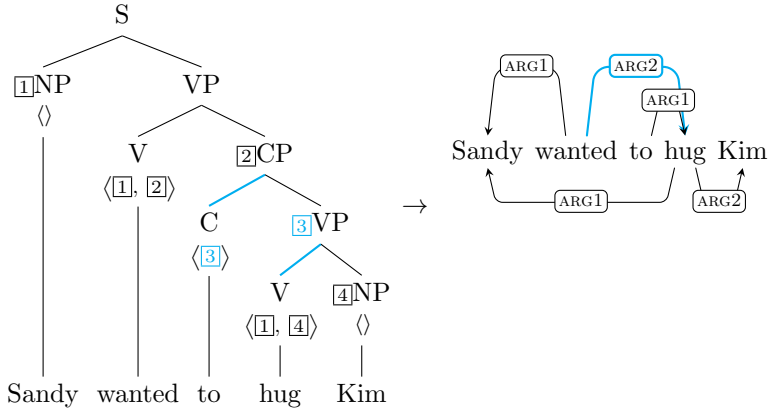
Another problem that arises is that content words often take functional phrases as arguments. This inflates the lengths of paths between content words, as shown in Ex. (2.76), which computationally practical dependency schemes try to avoid.

(2.76) Sandy wanted to hug Kim.

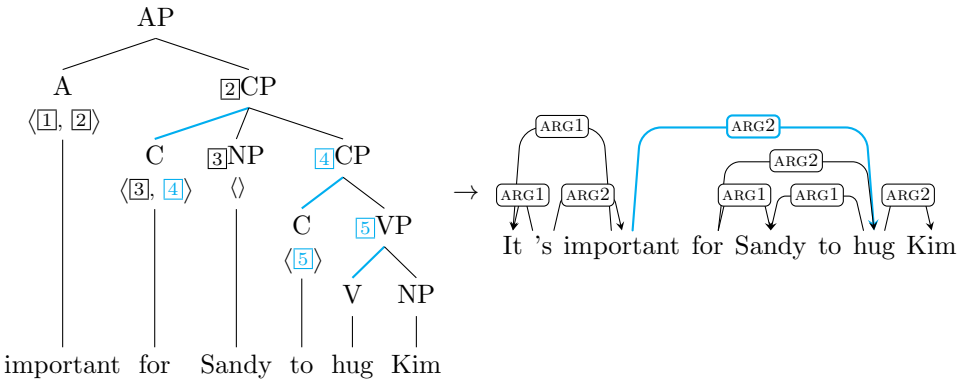


The representation can be brought a little bit closer to semantics and made a little bit more effective for computational purposes by forcing dependents to be content words rather than function words. Instead of linking words to their immediate arguments or (if their immediate arguments are phrasal) the head of their immediate arguments, words will be linked to the content word at the end of the chain of functional heads. When a function word has multiple arguments, the procedure will search for a content word in the last one. The new approach is shown in Exs. (2.77, 2.78).

(2.77) Sandy wanted to hug Kim.



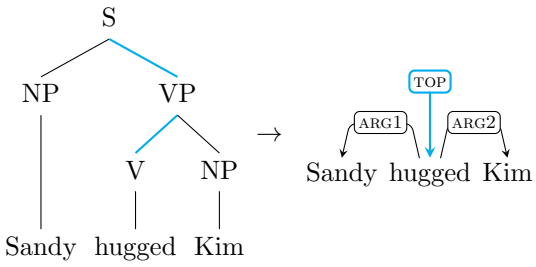
(2.78) It's important for Sandy to hug Kim.



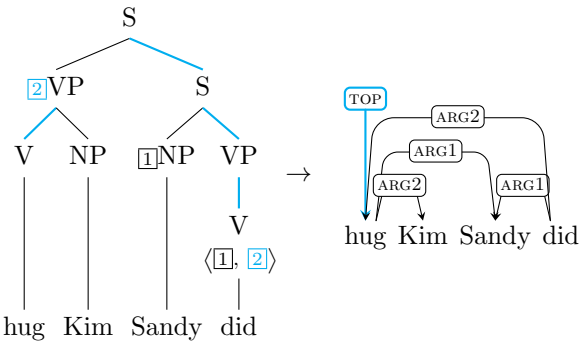
Head words and phrases are marked with bold edges for clarity. In Ex. (2.77), the word *wanted* would take *to* as dependent because it represents the first element on *wanted*'s argument structure list; however, because *to* is a function word, it can't be a dependent. Instead, *wanted* looks at the last element of *to*'s singleton argument structure list, and tries (successfully) to take that as a dependent. In Ex. (2.78), the adjective *important* takes a *for*-CP as argument. There is strong evidence that *for*-CPs have flat structure (Emonds, 1976; Sag, 1997), meaning that *for* has two elements on its argument structure list. By the heuristic laid out above, since *important* can't have *for* as a dependent, it looks to the second phrase on *for*'s argument structure list. The head of the second phrase, *to*, is also a function word, so the process keeps looking until it reaches *hug*.

The root of the dependency structure can be described similarly. The main verb of the sentence can be found by starting at the S node at the top of the tree, and following the sequence of syntactic heads down to a leaf node. If the leaf is a content word, then it gets marked as the TOP; if the leaf is a function word, then the first content word on the chain of functional argument structures is the TOP.

(2.79) Sandy hugged Kim.



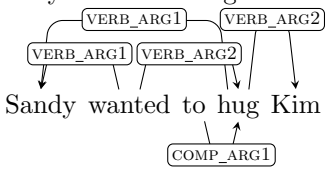
(2.80) ... hug Kim Sandy did.



The head word of the sentence in Ex. (2.79) is a content word, so it gets marked as the top of the sentence. But the head word of Ex. (2.80) is an auxiliary, so the word that heads its second argument is marked as the top instead. Thus a dependency formalism can be derived from HPSG's deeper syntactic structure by making every word the head of all elements on its original argument structure list, with the constraint that only content words can be dependents.

One limitation of this scheme is that the dependency label set is very restricted, containing only ARG1 through ARG4. In order to enrich the labeling scheme, the category of the head word can be prefixed to the ARG. This distinguishes verbal ARG1, which dictates a predicate's most agent-like argument, from adjectival ARG1, which indicates which phrase the predicate is modifying.

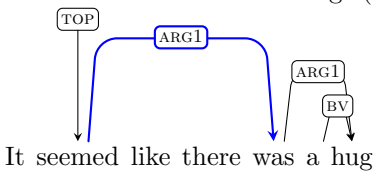
(2.81) Sandy wanted to hug Kim.



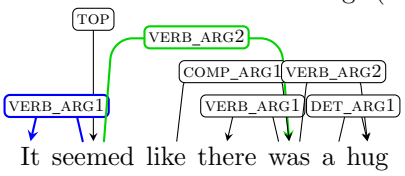
In Ex. (2.81), the verbs *wanted* and *hug* assign labels prefixed with VERB. The variant of HPSG that PAS originates from makes the non-standard assumption that *to* is a complementizer rather than an auxiliary verb. Thus *to* assigns the verb *hug* a label prefixed with COMP. This is the final change to the original HPSG representation that PAS makes.

The DM dependency representation is based on a pure semantic formalism, whereas the PAS representation is more closely related to a notion of argument structure, which contains syntactic information in addition to semantic. MRS and HPSG were developed side-by-side to complement each other, so DM and PAS bear many similarities. The main difference between their respective dependency formalizations is that PAS representation contains more semantically vacuous syntactic artifacts. In addition to function words like *to* getting their own dependents, semantically vacuous expletives are marked in PAS but excluded in DM. This has the minor downside of somewhat overloading the ARG1 label, which in PAS can refer to either a semantic agent or a semantic non-argument.

(2.82) It seemed like there was a hug. (DM)



(2.83) It seemed like there was a hug. (PAS)

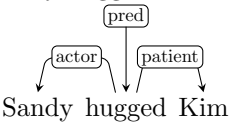


Because PAS includes relationships between content words and semantically vacuous words such as function words and expletives, every word in the dependency structure has at least one incoming or outgoing dependency edge. This means every PAS structure will be a *weakly connected* directed acyclic graph, in contrast to the tree-structured basic UD or the unconnected DM. It also strikes more of a balance between syntactic structure and semantic structure by marking the structure of syntactically necessary function words (at the cost of some semantic transparency in the case of expletives).

2.6 From Functional Generative Description to Prague Semantic Dependencies

The final dependency system discussed in this chapter is based on a syntactic theory known as *Functional Generative Description* (FGD; Sgall et al. 1986), which was developed with focus on Czech, a language that has far less rectrictive word order than English. If TG uses a constituency tree for both the surface and abstract syntactic representations, and LFG and HPSG use a constituency tree for the surface representation but a dependency-like graph for the abstract one, then FGD uses dependency graphs for both. The surface representation, also known as the *analytical* layer, contains mostly constituent-level information, with some simple functional labels like *subject* and *object*. The latent representation, known as the *tectogrammatical* layer (from Latin *tēctum*, "roof"), bears similarities to early Tranformational Grammar's deep syntactic structure, described in Section 2.2. It aims to represent the semantic structure of a sentence using the same data structure as the syntactic parse and it includes (coindexed when possible) empty tokens standing in for missing or disclocated arguments. Because the tectogrammatical representation is a dependency structure rather than a constituency structure, words can be explicitly labeled with semantic roles such as ACTOR and PATIENT.

(2.84) Sandy hugged Kim.



Linearization in the tectogrammatical structure is done according to two principles: a head and its dependents must be ordered according to their degree of topicality, with more topical words

occurring before more focal ones; and the result must be a dependency tree with no crossing branches. Topic and focus can be determined by deciding what question a sentence could be an answer to. The answer that fills the *wh* word is the focus, and words repeated from this question form the topic. In English, topic and focus information is typically indicated by intonation. Some examples are shown in Ex. (2.85–2.88), with boldface indicating which word has primary stress and with small caps indicating which one has secondary stress, if any.

(2.85) *What happened?*

SANDY hugged **Kim**
FOC FOC FOC

(2.86) *What did Sandy do?*

Sandy HUGGED **Kim**
TOP FOC FOC

(2.87) *Who did Sandy hug?*

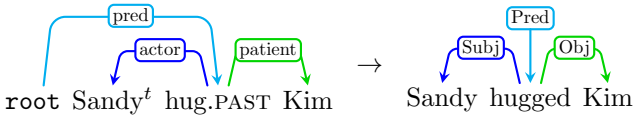
Sandy hugged **Kim**
TOP TOP FOC

(2.88) *What happened to Kim?*

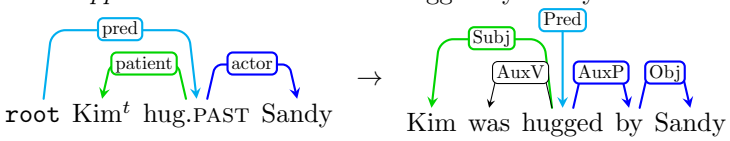
SANDY **hugged** Kim
FOC FOC TOP

In FGD's tectogrammatical representation, heads are linearized after topical dependents and before focal ones. This imposes a strict linear order on the tectogrammatical structure. Topic and focus are sometimes indicated in FGD by the presence or absence of a *t* (topic) superscript. While topic and focus in English are normally indicated by intonationally emphasizing the focus in a sentence with canonical word order, some marked word orders can achieve the same end. These non-canonical orders can be seen as reflecting the underlying tectogrammatical structure.

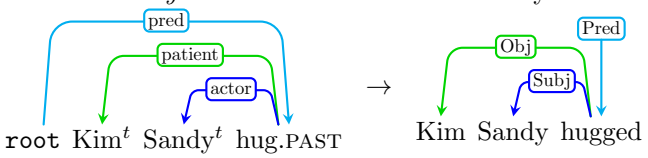
(2.89) *What did Sandy do?* Sandy hugged Kim.



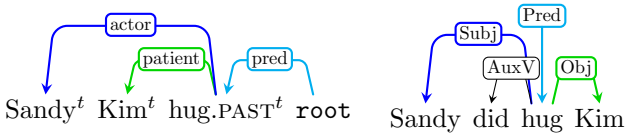
(2.90) *What happened to Kim?* Kim was hugged by Sandy.



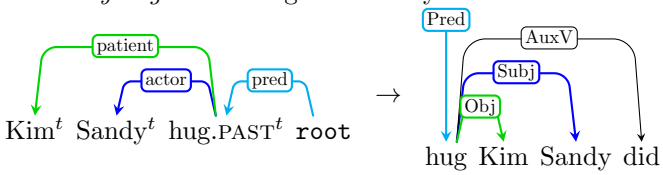
(2.91) *What did Sandy do Alex and Kim?* Alex Sandy waved at, but Kim Sandy hug.



(2.92) *Did Sandy hug Kim?* Sandy did hug Kim.



(2.93) *Did Sandy hug Kim?* Hug Kim Sandy did.

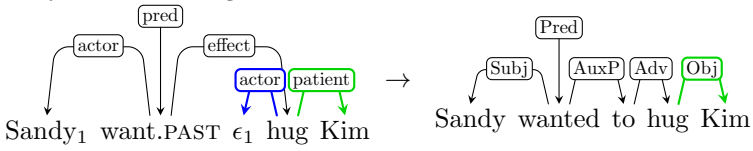


In Ex. (2.89), the sentence is shown with its canonical word order and an unmarked information structure. In Ex. (2.90), the patient of the predicate is topicalized, with the agent demoted to focus; this is expressed syntactically with the passive voice. Ex. (2.91) shows a sentence where both arguments of the verb are topics. This information structure permits noun phrase topicalization, where a non-subject argument is fronted to the beginning of the clause. Ex. (2.92, 2.93) show a sentence where all words are topics. Since the verb has two topical arguments, there are two possible tectogrammatical orderings—one with the agent first, and one with the patient first. These can be understood as the underlying representations for two distinct marked word orders: one where an emphatic auxiliary is added, and one where the verb and its object are additionally fronted. Note that the order of the subject and object with respect to each other in these examples remains constant when converting the tectogrammatical representation into the surface one.

Another feature of the tectogrammatical structure that can be seen in Ex. (2.89–2.93) is that function words are generally not represented in the abstract representation. In Ex. (2.90), the passive auxiliary and the preposition *by* were added according to the passivization transformation. Similary, the emphatic auxiliary *do* was inserted in Ex. (2.92, 2.93) via transformational rules. The exclusion of semantically vacuous function words reflects the goal of the tectogrammatical structure being a more abstract semantic representation of the sentence, like Minimal Recursion Semantics' semantic frame representation but unlike LFG's functional structure or HPSG's argument structure.

Control structures are handled by inserting a coindexed empty token into the tectogrammatical structure, much like TG's deep structure. These empty tokens are guaranteed to lack dependents, so they can be deleted in the surface structure.

(2.94) Sandy wanted to hug Kim.



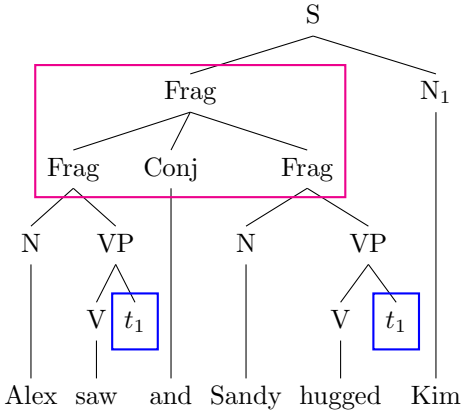
This analysis ensure a one-to-one mapping of tokens in the sentence to tokens in the dependency tree.

Coordination always poses a challenge in phrase structure grammars because groups of words that are generally not assumed to be phrasal constutents—such as *Alex saw* and *Sandy hugged* in Ex. (2.95)—can coordinate.

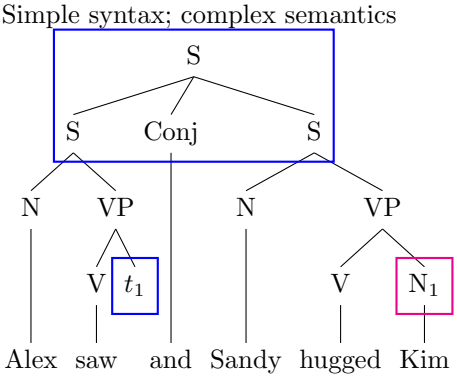
(2.95) Alex saw and Sandy hugged Kim.

Analyzing this sentence in terms of phrase structure is a daunting task. One option (Ex. 2.96a) involves stipulating construction-specific rules and categories in order to ensure that the common element *Kim* composes with both clausal fragments; this makes the syntax complicated in order to simplify the rules of semantic compositionality. The other option (Ex. 2.96b) involves explicitly representing the argument in one clause but not the other; this makes the syntax straightforward, but resolving the dependency between the first predicate and the second predicate's object requires its own construction-specific mechanisms.

(2.96) a. Complex syntax; simple semantics



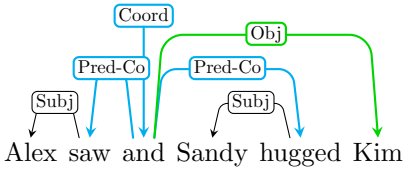
b. Simple syntax; complex semantics



In either case, phrase structure grammars are forced to distinguish between coordination that follows the phrase structure rules in the grammar and coordination that violates them. In the dependency-based FGD, on the other hand, coordination can be done relatively straightforwardly between phrases that are typically thought of as constituents as well as those that aren't. First, the conjunction is made to be the head, and all conjuncts are dependents of it. Second, any phrases common to all

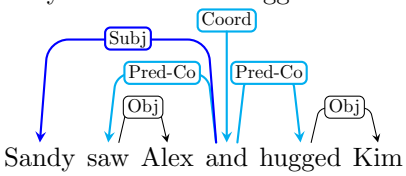
conjuncts are likewise made to be dependents of the conjunction. Finally, in order to distinguish between conjuncts and shared dependents, the actual conjuncts are marked as such. A surface-structure example is in Ex. (2.97).

(2.97) Sandy saw Alex and hugged Kim.



Because of the absence of additional phrasal structure, this approach can be applied just as easily to more canonical instances of coordination (Ex. 2.98).

(2.98) Sandy saw Alex and hugged Kim.

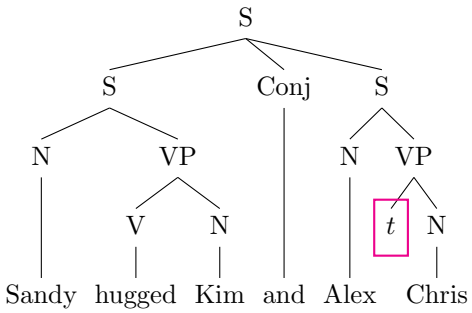


This is essentially the same as the "complex syntax, simple semantics" phrase structure approach, but without being forced to distinguish between two different types of coordination. The analysis presented in Exs. (2.97, 2.98) is typically applied to both the surface and deep representations in FGD.

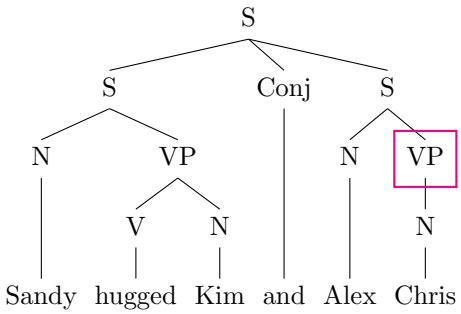
If coordination is one place where dependency representations appear to shine, then the related ellipsis phenomenon known as *gapping* is one place where they struggle. This was mentioned briefly in Section 2.1.2 but will be discussed here in more detail. In gapped constructions, two or more sets of arguments for a predicate occur in multiple conjoined clauses, but the predicate is only overtly present in the first one. The phrase structure approach again has options: build the complete phrase structure and delete the repeated verb as a kind of post-processing (which would be preferred in TG), allow a verb phrase without an overt head (which would be preferred in LFG), or stipulate another fragmental phrase structure category (which would be preferred in HPSG).

(2.99) Sandy hugged Kim, and Alex [hugged] Chris.

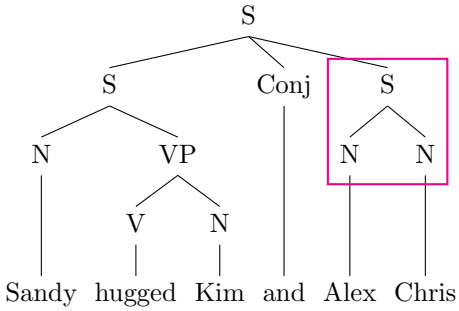
a. Empty token



b. Headless VP

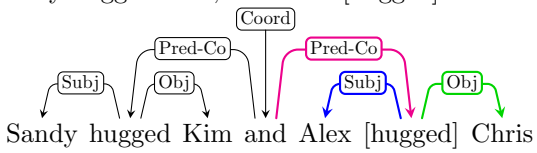


c. Fragment



A dependency approach, on the other hand, is forced into one option. Because arguments depend on their head, but the head in the second clause is missing, there's simply nowhere for them to go. The only way to handle this in FGD is to stipulate a silent node in the second clause at both tectogrammatical structure and surface structure.

(2.100) Sandy hugged Kim, and Alex [hugged] Chris.

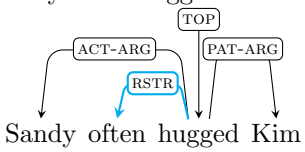


This is most analogous to the deletion approach in a phrase structure grammar. While silent nodes are needed in the tectogrammatical structure for the same reason they're needed in TG's deep structure, until this point they've been omissable from the surface structure. Consequently, the

one-to-one mapping of tokens in the surface tree and tokens in the sentence that can be maintained for all other phenomena is necessarily violated solely for this one construction.

Since the tectogrammatical representation is meant to be a semantic representation, and is already dependency-structured, it was converted into a semantic dependency formalism just like MRS and HPSG. The semantic dependency version of FGD is known as Prague Semantic Dependencies (PSD; Hajic et al. 2012; Oepen et al. 2014). The tectogrammatical representation is a projective tree, and as a consequence the Prague Semantic Dependencies scheme is mostly tree-like, with only a few exceptions to be discussed shortly. In order to have general-purpose utility, any dependency annotation scheme needs to maintain the linear order of words in the sentence. Consequently, the topic and focus information cannot be represented in the same way that the original framework represents it. Additionally, annotating topic and focus in a large corpus is very labor-intensive, and isn't currently represented in the Prague Czech-English Dependency Treebank (Hajic et al., 2012), the largest dependency treebank annotated with the FGD framework. Thus PSD lacks any topic or focus information, either through word order or other annotations. The PSD dependency labels are almost identical to the FGD tectogrammatical labels, with the exception that argument labels (ACT, PAT, EFF) are explicitly marked as being arguments. Like UD, but unlike DM or PAS, modifiers are dependents of the words they modify.

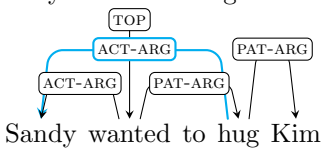
(2.101) Sandy often hugged Kim.



In Ex. (2.101), the main predicate is marked as the root of the tree (TOP), the agent and patient are labeled as such, and the adverb *often* is a dependent of the predicate.

As with UD+ and the previously described semantic dependency formalisms, PSD annotates edges in contrustructions where a predicate's subject is located elsewhere.

(2.102) Sandy wanted to hug Kim.

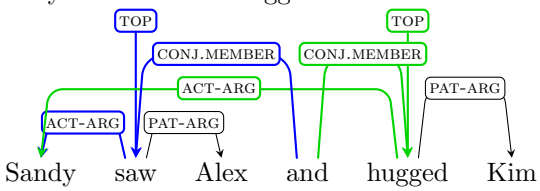


In the original FGD tectogrammatical tree, *Sandy* would only be coindexed with an empty token that depends on the lower predicate, similar to TG. In PSD, these coindexical relationships are explicitly represented as edges.

Coordination constructions likewise introduce extra edges into the dependency graph. The conjuncts are all made to be dependents on the conjunction, as in the tectogrammatical structure, but

they also all depend on the word that acted as the head of the conjunction in the original representation. Similarly, any non-conjunct dependents of the original conjunction are instead attached to each of the conjuncts.

(2.103) Sandy saw Alex and hugged Kim.



In Ex. (2.103), the two predicates *saw* and *hugged* are conjoined by *and*. In addition to being explicitly marked as such, they also share the *top* relation and they share their ACT-ARG dependent. Gapped constructions, however, are intentionally unrepresented. Sentences with gaps were removed from all three SemEval datasets—DM, PAS, and PSD—because of how hard it is to analyze them. Annotating them sensibly in a dependency formalism typically requires re-generating the missing predicate, which most popular statistical parsing algorithms aren't designed for. There has been recent work on finding the "least bad" solution to this (Schuster et al., 2018), but at the moment it remains an open problem.

2.7 Conclusion

This chapter has discussed the theoretical foundations of the most well-known syntactic representations. First it showed some evidence for constituency structure in language, introducing the reader to the kinds of tasks theorists set out to accomplish. Then it described the strengths and weaknesses of the two main ways of describing the kind of easily testable surface-level syntactic structure, pointing out that phrase structure tends to be more restrictive than dependency structure. The rest of the chapter described five different syntactic representations, each coming from a different theory of underlying syntax.

Because several of the dependency annotation schemes have lost mechanisms important to the original theories, they all carry more or less the same information about argument structure, with only subtle differences between them. These differences are summarized in Table 2.1. Other than having a different theoretical origin, PTB and UD differ primarily in that UD is a dependency framework with functional labels and PTB is a traditional phrase structure scheme. PTB mostly utilizes categorial labels, with limited functional information, and UD utilizes functional labels, with limited categorial information. UD+ simply extends UD to permit multiple heads with no restrictions on cyclicity. UD+, DM, PAS, and PSD all allow for multiple heads; however, they can be uniquely categorized along two dimensions. Firstly, UD+ and PSD both derive as relaxed extensions of strict tree structures, and remain mostly tree-like; PAS and DM, on the other hand, come from

Scheme	Theory	Dep	Acyclic	>1 parent	Conn.	Tree-like	Labels
PTB	TG	✗	✓	✗	✓	✓	Category/ (Function)
UD	LFG	✓	✓	✗	✓	✓	Function/ (Category)
UD+	LFG	✓	✗	✓	✓	✓	Function/ (Category)
PAS	HPSG	✓	✓	✓	✓	✗	Category/ Protorole
PSD	FGD	✓	✓	✓	✗	✓	Semantic role
DM	MRS	✓	✓	✓	✗	✗	Protorole

Table 2.1: Differences between syntactic representations.

representations that never enforced tree structure, making them more graph-like. Secondly, UD+ and PAS draw inspiration from theoretical structures that aim to interface between surface syntax and semantics. In part because of this, even semantically vacuous function words—which are critical for converting a semantic denotation into a valid sentence—are assigned a role in the dependency graph, meaning the formalisms are *weakly connected* graphs. PSD and DM, on the other hand, aim to be coherent semantic representations. There's simply no place for function words in these formalisms, so PSD and DM exclude them from their dependency graphs. Each of these four representations has its own way of labeling the edges, but they all boil down to marking semantic roles of varying levels of granularity. While one could argue that UD+ is only committed to providing *functional* rather than *semantic* labels, the only place where the distinction is clearly meaningful in practice is in passive constructions—but even still, most treebanks adopt an extension that explicitly marks passive subjects and auxiliaries, meaning that the semantic roles are recoverable from the functional labels. This means that a system with the capacity to learn one of these formalisms should, in principle, have the capacity to learn the others.

One takeaway from this discussion is that the distinction between "syntactic" and "semantic" formalisms is at best fuzzy and perhaps nonexistent. If anything, there's a syntactico-semantic *continuum*, with the different formalisms falling on different points on that spectrum. PTB encodes primarily surface-level constituency information, making it the most "syntactic" of the formalisms. UD aims to encode a kind of "deep structure" instead, making it closer to semantics, but it sacrifices some critical semantic relationships to retain a strict tree structure reminiscent of surface-level phrase structure trees. UD+ relaxes the strict tree structure in order to bring that missing information into the formalism, making it more semantic than basic UD. PAS and PSD both have at least one syntactic feature in common with UD (connectedness and tree-like structure, respectively), but adopt additional semantic features as well. DM is the most semantic formalism, drawing more on semantic theory than all other frameworks discussed here. Labeling the formalisms binarily as either "syntactic" or "semantic" is not clearly useful. UD+ could, for instance, trivially increase

how "semantic" it is by pruning vacuous function words or reversing the direction of edges between modifiers and their predicates. It wouldn't need to actually add any more information, because the alleged semantic formalisms don't encode much semantic information beyond the same argument selection and predicate modification already present in Universal Dependencies.

While it's good to keep in mind the different theoretical origins of these frameworks and how they are reflected in the representations, a more illuminating distinction in the present context of statistical parsing might be between strictly tree-structured and graph-structured formalisms. This could be useful to keep in mind when choosing which formalisms to use in a downstream application. Parsers for and downstream users of strictly tree-structured formalisms can take advantage of the structural constraints to simplify the parsing algorithm, whereas parsers for graph-structured formalisms need to use additional tricks to capture all and only correct edges in the relatively sparse representations. It would be reasonable to train a system that works for the "semantic" dependency formalisms on the "syntactic" UD+ formalism (provided it can generate cyclical graphs). However, it wouldn't be nearly as reasonable to train a system that works for the graph-structured dependency formalisms on the strictly tree-structured dependency formalism basic UD, because it would fail to take advantage of the relative simplicity the representation. Conveniently, this proposed distinction partitions the formalisms along the syntactico-semantic continuum; the more syntactic PTB and UD are both strictly tree-structured (under the assumption that coindexation doesn't violate tree structure), whereas the more semantic UD+ and SemEval dependency formalisms are all graph-structured.

Chapters 5 and 6 will motivate and extend a neural network parser for producing strictly tree-structured dependency formalisms. Tree-structured dependency formalisms are the most popular, likely owing to their simplicity and user-friendliness. As such, Chapter 5 establishes a baseline system for dependency trees, and emphasizes its performance on English Stanford Dependencies (though it is evaluated on a number of other tree-structured datasets). Chapter 6 extends it to Universal Dependencies, eventually including techniques to generate the Universal Features that are a critical part of LFG but were removed during its simplification to Stanford Dependencies. Chapter 7 will show how this parser can be minimally extended to produce arbitrary graph-structured formalisms. Specifically, the parser can be tweaked slightly to allow it to generate the more complex graph-structured dependency formalisms inspired by the other linguistic frameworks.

Chapter 3

Machine Learning

3.1 Affine classification

3.1.1 Naïve Bayes and Maximum Entropy Classifiers

Many high-level machine learning tasks can be construed as involving some sort of classification objective. This section will derive the affine softmax classifier mathematically from the simple case of classifying a set of features. Subsequent discussions will build off ideas motivated here to show that *bi*affine softmax classifiers can be mathematically derived for some special cases, including dependency parsing and dependency labeling.

For notational clarity, variables will be indexed by i and t when relevant, where i indicates that it comes from the i-th example in the treebank and t indicates it is the t-th timestep in that example. The notation will follow the convention of using lowercase italics for scalar variables, lowercase bold for vectors, uppercase italics for matrices, and uppercase bold for higher order tensors. This will be maintained when indexing, so that $\mathbf{a}_i$ is the ith row of A. When indexing into a function that returns a non-scalar, the index will come before the function's arguments.

$$A = \text{stack}(\mathbf{a}_1, \ldots, \mathbf{a}_n) \tag{3.1}$$

$$\mathbf{a}_i = \text{unstack}_i(A) \tag{3.2}$$

The only exception to this will be that capital letters will also occasionally be used for the lengths of sequences when the , as in $(a_1, \ldots, a_t, \ldots, a_T)$.

Consider a task that involves classifying new emails into one of three classes—`spam`, `not spam`, and `urgent`—based on a training corpus of eamils whose classes are already known. An engineer might identify a handful of binary features that could be predictively correlated with the different classes. These boolean features could be things like words that occur in the bodies of the training

emails (e.g. `in_body:free`), words that occur in the headers of the emails (`in_header:FW`), or the domains of the sender email addresses (`sender_domain:@stanford.edu`). Let each class be $c_k \in \mathbf{c} = (c_1, \ldots, c_m)$, and let each feature be $f_j \in \mathbf{f} = (f_1, \ldots, f_d)$. $\mathbf{c}$ will be categorical, meaning exactly one c_k will take the value 1 for each example i and the rest will be 0, and f_j will be binary, taking values of 0 or 1. Each training example i will have a known class y_{ik} that provides a value for variable c_k, and it will also have a known feature set $\mathbf{x}_i$ that provides the value for variable $\mathbf{f}$. So if c_1 represents `spam`, and example i is known to be a spam email, then $\mathbf{y}_i = (1, 0, 0)$; and if f_1 represents `in_body:free`, and exmple i contains the word *free* in its body text, then $x_{ij} = 1$. The engineer's goal now is to design an algorithm that will compute the probability $P(c_k = 1|\mathbf{f} = \mathbf{x}_i)$ for each class c_k and for each new incoming email i. Observation i is then assumed to be a member of the class with the highest probability. The engineer needs to find a way to determine how predictive each feature is of the three classes so that they can compute the probability of a new email with a new set of features falling into each of the three classes. How should they go about this?

The conditional probability of a class given the features is the ratio between the joint probability of the observation and the class $\mathbf{f} = \mathbf{x}_i, c_k = 1$ and the marginal probability of just the observation $\mathbf{f} = \mathbf{x}_i$ (Eq. 3.3). In turn, the probability of an observation—a set of feature-value pairs—is the joint probability of every feature in the observation (Eq. 3.4). Similarly, the probability of a set of feature-value pairs and a class c_k is the probability of the observation and the class together (Eq. 3.5).

$$P(c_k = 1|\mathbf{f} = x_i) = \frac{P(\mathbf{f} = \mathbf{x}_i, c_k = 1)}{P(\mathbf{f} = \mathbf{x}_i)} \tag{3.3}$$

$$P(\mathbf{f} = \mathbf{x}_i) = P(f_1 = x_{i1}, \ldots, f_d = x_{id}) \tag{3.4}$$

$$P(\mathbf{f} = \mathbf{x}_i, c_k = 1) = P(f_1 = x_{i1}, \ldots, f_d = x_{id}, c_k = 1) \tag{3.5}$$

Computing the probability in Eq. (3.3) requires doing two things: first, expressing the probability as a function of a fixed number of free parameters; and second, optimizing the parameters for a labeled training set. The expressions above problematically treat feature vectors holistically, with no internal structure and with no relationship to nearly identical feature sets. Since each individual feature vector $\mathbf{x}_i$ will likely be extremely rare in a training corpus, accurately estimating the parameters for these probabilities with a limited amount of training data will be extremely difficult. That is, every possible combination of features will need to receive its own free parameter, making it likely that there will be more parameters than observations. This problem of similar feature sets having no relationship to each other can be addressed by making some simplifying assumptions about the distributions of the individual features. First Eq. (3.3) will need to be rewritten using Bayes' rule. Eq. (3.6–3.8) show how to derive and apply Bayes' rule in the probability under consideration. Here,

the variable $c_k = 1$ will be expressed as c_k, and likewise $f_j = x_{ij}$ will be shortened to simply f_j.

$$P(c_k|\mathbf{f}) = \frac{P(f_1, \ldots, f_d, c_k)}{P(f_1, \ldots, f_d)} \tag{3.6}$$

$$= \frac{P(f_1, \ldots, f_d, c_k)\,\frac{P(c_k)}{P(c_k)}}{P(f_1, \ldots, f_d)} \tag{3.7}$$

$$= \frac{P(f_1, \ldots, f_d|c_k)P(c_k)}{P(f_1, \ldots, f_d)} \tag{3.8}$$

Eq. (3.6) expands the lefthand side by the definition of conditional probability. Eq. (3.7) multiplies the numerator by $P(c_k)/P(c_k)$. Eq. (3.8) then contracts the numerator by the definition of conditional probability. In this equation, the conditional probability $P(c_k|\mathbf{f})$ is also known as the *posterior*; the marginal probability $P(c_k)$ is also known as the *prior*; and the conditional probability $P(\mathbf{f}|c_k)$ is also known as the *likelihood*. In order to make the posterior probability tractable to estimate from a finite set of training data, the probabilities conditioned on unbounded numbers of features (e.g. $f_j, \ldots, f_d$) need to instead be conditioned on a fixed number of features. This can be done by assuming conditional independence (Eq. 3.9) between the features.

$$P(x_1, x_2|x_3) = P(x_1|x_3)P(x_2|x_3) \qquad \text{Conditional Independence} \tag{3.9}$$

$$P(c_k|\mathbf{f}) = \frac{P(f_1, \ldots, f_d|c_k)\,P(c_k)}{P(f_1, \ldots, f_d)} = \frac{\prod_{j=1}^{d}\left[P(f_j|c_k)\right]P(c_k)}{P(f_1, \ldots, f_d)} \tag{3.10}$$

While this solves the problem in the numerator, the denominator of Eq. (3.10) is still the probability of an unbounded number of variables. There are two ways it can be simplified. The first is to additionally stipulate *mutual* independence (Eq. 3.11) between the feature probabilities.

$$P(z_1, z_2, z_3) = P(z_1)P(z_2)P(z_3) \qquad \text{Mutual Independence} \tag{3.11}$$

$$P(c_k|\mathbf{f}) = \frac{\prod_{j=1}^{d}\left[P(f_j|c_k)\right]P(c_k)}{P(f_1, \ldots, f_d)} = \frac{\prod_{j=1}^{d}\left[P(f_j|c_k)\right]P(c_k)}{\prod_{j=1}^{d}\left[P(f_j)\right]} \tag{3.12}$$

$$= \prod_{j=1}^{d}\left[\frac{P(f_j|c_k)}{P(f_j)}\right]P(c_k) \tag{3.13}$$

The large probability in the denominator is broken up into a product of marginal probabilities (Eq. 3.12), which can then be factored into the likelihood (Eq. 3.13). However, if mutual independence doesn't hold exactly, then the posterior won't be a valid probability distribution that sums to one.

Instead, it can be easily seen to sum to the ratio $P(f_1, \ldots, f_d) / \prod_{j=1}^{m} P(f_j)$. This leads to the second way that Eq. (3.10) can be simplified, which rewrites the denominator without making any additional assumptions.

$$P(c_k|\mathbf{f}) = \frac{\prod_{j=1}^{d} [P(f_j|c_k)]\, P(c_k)}{P(f_1, \ldots, f_d)} = \frac{\prod_{j=1}^{d} [P(f_j|c_k)]\, P(c_k)}{\sum_{k=1}^{m} P(f_1, \ldots, f_d, c_k)} \tag{3.14}$$

$$= \frac{\prod_{j=1}^{d} [P(f_j|c_k)]\, P(c_k)}{\sum_{k=1}^{m} P(f_1, \ldots, f_d, c_k) \frac{P(c_k)}{P(c_k)}} \tag{3.15}$$

$$= \frac{\prod_{j=1}^{d} [P(f_j|c_k)]\, P(c_k)}{\sum_{k=1}^{m} P(f_1, \ldots, f_d | c_k) P(c_k)} \tag{3.16}$$

$$= \frac{\prod_{j=1}^{d} [P(f_j|c_k)]\, P(c_k)}{\sum_{k=1}^{m} \prod_{j=1}^{m} [P(f_j|c_k)]\, P(c_k)} \tag{3.17}$$

$$= \frac{\prod_{j=1}^{d} \left[\frac{P(f_j|c_k)}{P(f_j)} \right] P(c_k)}{\sum_{k'=1}^{m} \prod_{j=1}^{m} \left[\frac{P(f_j|c_{k'})}{P(f_j)} \right] P(c_{k'})} \tag{3.18}$$

Eq. (3.14) introduces a new variable c_k into the probability of the denominator, but then sums over it, maintaining exact equivalence. Eqs. (3.15, 3.16) then multiply the denominator by $P(c_k)/P(c_k)$ and create a conditional probability given c_k, just as Bayes' Rule does (cf. Eqs. 3.7, 3.8). Using the assumption of conditional independence already assumed in Eq. (3.10), the conditional probability in the denominator can be split into a product of conditional probabilities (Eq. 3.17). The numerator and denominator can then both be multiplied by $\prod_{j=1}^{d} P(f_j)$, making the expression maximally comparable to Eq. (3.13). In fact, the only difference between the version that assumes both conditional independence and mutual independence (Eq. 3.19) and the version that only assumes conditional independence (Eq. 3.20) is that the former doesn't need explicit normalization,

whereas the latter does.

$$P^{(\text{NB})}(c_k|\mathbf{f}) = \prod_{j=1}^{d}\left[\frac{P(f_j|c_k)}{P(f_j)}\right]P(c_k) \qquad \text{conditional\&mutual ID} \tag{3.19}$$

$$P^{(\text{ME})}(c_k|\mathbf{f}) = \frac{\prod_{j=1}^{d}\left[\frac{P(f_j|c_k)}{P(f_j)}\right]P(c_k)}{\sum_{k'=1}^{m}\prod_{j=1}^{d}\left[\frac{P(f_j|c_{k'})}{P(f_j)}\right]P(c_{k'})} \qquad \text{conditional ID only} \tag{3.20}$$

On the surface, this may seem trivial, since the normalization term scales the whole probability distribution proportionally, and won't change which class c_k has the highest probability. However, it will be shown shortly that in fact this difference has critical implications for finding the optimal parameters for probabilities $P(f_j|c_k), P(f_j), P(c_k)$. In particular, Eq. (3.19) is the expression for a *Naïve Bayes* (NB) classifier, for which the optimal probabilites can be estimated from a training corpus by simply counting frequences; Eq. (3.20), on the other hand, is the expression for a *maximum entropy* (ME) classifier, which has no analytic optimum.

The expressions for the two classifiers will be easier to work with if they're represented with sums of log-probabilities rather than products of raw probabilities. The Naïve Bayes and Maximum Entropy classifiers can be expressed as different functions of the un-normalized log-probability—or *score*—s_{ik} of the class c_k given the feature set in example i (Eq. 3.23). Specifically, NB applies the exponential function to the score (Eq. 3.24), whereas ME applies the softmax function (Eqs. 3.21, 3.25).

$$\text{softmax}_k(\mathbf{x}) = \frac{\exp(x_k)}{\sum_{k'=1}^{m}\exp(x_{k'})} \tag{3.21}$$

$$s_{ik} = \ln\left(\prod_{j=1}^{d}\left[\frac{P(f_j|c_k)}{P(f_j)}\right]P(c_k)\right) \tag{3.22}$$

$$= \sum_{j=1}^{d}\left[\ln\left(\frac{P(f_j|c_k)}{P(f_j)}\right)\right] + \ln(P(c_k)) \tag{3.23}$$

$$P^{(\text{NB})}(c_k|\mathbf{f}) = \exp_k(\mathbf{s}_i) \tag{3.24}$$

$$P^{(\text{ME})}(c_k|\mathbf{f}) = \text{softmax}_k(\mathbf{s}_i) \tag{3.25}$$

Because the ME classifier involves applying the softmax function to the score, it is also commonly known as a *softmax regression* classifier.

The score s_{ik} in Eq. (3.23) is expressed in terms of three probabilites $P(f_j|c_k), P(f_j), P(c_k)$. In order to learn a classifier from training data, the score needs to be expressed in terms of parameters

that can be optimized. The most natural probability distribution for the binary-valued variables in the likelihood is the *Bernoulli* distribution, which is characterized by a single parameter p that can be interpreted as the probability of the variable taking the value 1. The class variable c_k is m-ary rather than binary, meaning the categorical distribution is ideal for modeling the prior.[1]

$$\text{Bernoulli}(x; p) = p^x (1-p)^{1-x} \tag{3.26}$$

$$\text{Categorical}(x; \mathbf{p}) = \prod_{k=1}^{m} \left[p_k^{\{x=k\}} \right] = p_x \tag{3.27}$$

$$P\left(f_j = x_{ij}; p_j^{(f)}\right) = \text{Bernoulli}\left(x_{ij}; p_j^{(f)}\right) \tag{3.28}$$

$$P\left(f_j = x_{ij} | c_k; p_{kj}^{(fc)}\right) = \text{Bernoulli}\left(x_{ij}; p_{kj}^{(fc)}\right) \tag{3.29}$$

$$P\left(c_k = 1; \mathbf{p}^{(c)}\right) = \text{Categorical}\left(k; p_k^{(c)}\right) \tag{3.30}$$

Notationally, the parameters for $P(f_j)$ are superscripted with (f), the parameters for $P(f_j|c_k)$ are superscripted with (fc), and the parameters for $P(c_k)$ are superscripted with (c). θ will be used as shorthand for all of $P^{(fc)}, \mathbf{p}^{(f)}, \mathbf{p}^{(c)}$. These probability mass functions can now be substituted for the probabilities in the score from Eq. (3.23), shown in Eq. (3.33).

$$s_{ik} = \sum_{j=1}^{d} \left[\ln\left(\frac{P(f_j|c_k)}{P(f_j)} \right) \right] + \ln(P(c_k)) \tag{3.31}$$

$$= \sum_{j=1}^{d} \left[\ln\left(\frac{\left(p_{kj}^{(fc)}\right)^{x_{ij}} \left(1 - p_{kj}^{(fc)}\right)^{1-x_{ij}}}{\left(p_j^{(f)}\right)^{x_{ij}} \left(1 - p_j^{(f)}\right)^{1-x_{ij}}} \right) \right] + \ln\left(p_k^{(c)}\right) \tag{3.32}$$

$$s_{ik} = \sum_{j=1}^{d} \left[x_{ij} \ln\left(\frac{p_{kj}^{(fc)}}{p_j^{(f)}} \right) + (1 - x_{ij}) \ln\left(\frac{1 - p_{kj}^{(fc)}}{1 - p_j^{(f)}} \right) \right] + \ln\left(p_k^{(c)}\right) \tag{3.33}$$

With the vector of scores $\mathbf{s}_i$ now parameterized, its optimal parameters can be computed from a labeled training corpus.

The "optimal parameters" can be defined as the ones that maximize the probability of the data observed in the training corpus X, Y (Eq. 3.35). These are known as the *maximum likelihood estimators*. The maximum likelihood estimators also minimize the *cross-entropy* of the data, which is the total negative log probability (negative log probability will be notated with $\mathcal{L}$) of all correct classes given each feature set. Since the variable $\mathbf{c}$ has already been assumed to follow a Categorical distribution, the Categorical distribution probability mass function can be used to represent the

[1] Let $\{x = i\}$ be the boolean equality operator, which evaluates to 1 iff x and i are equal and 0 iff they're different.

probability of a class given a feature set.

$$P(Y|X;\theta) = \prod_{i=1}^{n}\prod_{k=1}^{m}[P(c_k|\mathbf{f})^{y_{ik}}] \tag{3.34}$$

$$\mathcal{L}(Y|X;\theta) = -\sum_{i=1}^{n}\sum_{k=1}^{m}\left[y_{ik}\ln\left(P(c_k|\mathbf{f})\right)\right] \tag{3.35}$$

Summing each example over all possible classes for each example i but multiplying by y_{ik}—which is 0 for incorrect classes but 1 for the correct class—allows only the entropy of the correct class to aggregated into the sum. Substituting the NB un-normalized conditional probability for $P(c_k|\mathbf{f})$ yields Eq. (3.38).

$$\mathcal{L}(Y|X;\theta) = -\sum_{i=1}^{n}\sum_{k=1}^{m}\left[y_{ik}\ln\left(P^{(\mathrm{NB})}(c_k|\mathbf{f})\right)\right] \tag{3.36}$$

$$= -\sum_{i=1}^{n}\sum_{k=1}^{m}[y_{ik}\ln(\exp(s_{ik}))] \tag{3.37}$$

$$= -\sum_{i=1}^{n}\sum_{k=1}^{m}[y_{ik}s_{ik}] \tag{3.38}$$

This objective is convex, and can be optimized with respect to $p_{kj}^{(fc)}$ and $p_j^{(f)}$ by setting the gradient equal to zero. Solving for $p_k^{(c)}$ can be done by adding a lagrangian constraint $\lambda(\sum_{k=1}^{m}[p_k^{(c)}]-1)=0$ to ensure that the priors are a valid probability distribution. The gradients with respect to each parameter are provided in Eqs. (3.39–3.42).

$$0 = \nabla_{p_{kj}^{(fc)}}\mathcal{L}(Y|X;\theta) = -\sum_{i=1}^{n}\left[y_{ik}\frac{x_{ij}-p_{kj}^{(fc)}}{p_{kj}^{(fc)}(1-p_{kj}^{(fc)})}\right] \tag{3.39}$$

$$0 = \nabla_{p_j^{(f)}}\mathcal{L}(Y|X;\theta) = -\sum_{i=1}^{n}\left[-\frac{x_{ij}-p_j^{(f)}}{p_j^{(f)}(1-p_j^{(f)})}\right] \tag{3.40}$$

$$0 = \nabla_{p_k^{(c)}}\mathcal{L}(Y|X;\theta,\lambda) = -\sum_{i=1}^{n}\left[\frac{y_{ik}}{p_k^{(c)}}\right]+\lambda \tag{3.41}$$

$$0 = \nabla_{\lambda}\mathcal{L}(Y|X;\theta,\lambda) = \sum_{k=1}^{m}\left[p_k^{(c)}\right]-1 \tag{3.42}$$

The solutions to Eqs. (3.39–3.41) follow from basic algebra, and are given in Eqs. (3.43–3.45).

$$p_{kj}^{(fc)} = P(f_j = 1|c_k) = \frac{1}{\sum_{i=1}^{n} [y_{ik}]} \sum_{i=1}^{n} [x_{ij} y_{ik}] \tag{3.43}$$

$$p_j^{(f)} = P(f_j = 1) = \frac{1}{n} \sum_{i=1}^{n} [x_{ij}] \tag{3.44}$$

$$p_k^{(c)} = P(c_k = 1) = \frac{1}{n} \sum_{i=1}^{n} [y_{ik}] \tag{3.45}$$

These solutions are very simple; they show that the optimal NB classifier can be determined by just counting the number of times features and classes occur and co-occur. The analytical, frequency-based solutions here are only possible because of the mutual independence assumptions that allowed the normalization term to be ignored. However, if the posterior is normalized in a ME classifier to avoid the stronger independence assumptions, then no analytic solution is possible for $p_k^{(c)}$ and $p_{kj}^{(fc)}$.

$$\mathcal{L}(Y|X;\theta) = -\sum_{i=1}^{n}\sum_{k=1}^{m} \left[y_{ik} \ln \left(P^{(\mathrm{ME})}(c_k|\mathbf{f}) \right) \right] \tag{3.46}$$

$$= -\sum_{i=1}^{n}\sum_{k=1}^{m} \left[y_{ik} \ln \left(\frac{\exp(s_{ik})}{\sum_{k'=1}^{m} [\exp(s_{ik'})]} \right) \right] \tag{3.47}$$

$$= -\sum_{i=1}^{n}\sum_{k=1}^{m} \left[y_{ik} \left(\ln(\exp(s_{ik})) - \ln \left(\sum_{k'=1}^{m} [\exp(s_{ik'})] \right) \right) \right] \tag{3.48}$$

$$= -\sum_{i=1}^{n}\sum_{k=1}^{m} \left[y_{ik} \left(s_{ik} - \ln \left(\sum_{k'=1}^{m} [\exp(s_{ik'})] \right) \right) \right] \tag{3.49}$$

The log-normalization term in Eq. (3.49) is complex, resulting in substantially more complex gradients.

$$0 = \nabla_{p_{kj}^{(fc)}} \mathcal{L}(Y|X;\theta) = -\sum_{i=1}^{n} \left[y_{ik} \frac{x_{ij} - p_{kj}^{(fc)}}{p_{kj}^{(fc)}(1 - p_{kj}^{(fc)})} \left(1 - \mathrm{softmax}_k(\mathbf{s}_i)\right) \right] \tag{3.50}$$

$$0 = \nabla_{p_j^{(f)}} \mathcal{L}(Y|X;\theta) = -\sum_{i=1}^{n} \left[-\frac{x_{ij} - p_j^{(f)}}{p_j^{(f)}(1 - p_j^{(f)})} (m-1) \right] \tag{3.51}$$

$$0 = \nabla_{p_k^{(c)}} \mathcal{L}(Y|X;\theta) = -\sum_{i=1}^{n} \left[\frac{y_{ik}}{p_k^{(c)}} \left(1 - \mathrm{softmax}_k(\mathbf{s}_i)\right) \right] \tag{3.52}$$

The parameters $p_j^{(f)}$ have the same optimum in the normalized version as the un-normalized version, but the $(1 - \mathrm{softmax}_k)$ term in the gradients for the others contains all parameters θ, blocking them

from being isolated and solved. That is, for these other parameters, the gradient of the objective for each example i is weighted by how much probability mass the *current estimates* assign to incorrect classes. Consequently, these parameters have much more complex interdependence, and the gradients can't simply be solved for the parameters. Consequently, they must be estimated through iterative optimization techniques such as gradient descent. While the optimization is more complex, the learned parameters tend to be more robust in the presence of mutual dependence. For example, if a feature f_j has a redundant correlate $f_{j'}$ such that $f_j \Leftrightarrow f_{j'}$, then the estimates of the un-normalized NB version will overrepresent the impact of f_j on the posterior probability. This may reduce the probability of the observed class in some training examples. By contrast, in a ME model, the gradient will give more weight to these "more incorrect" training examples during optimization, which will lower the strength of features f_j and $f_{j'}$. In the case of mutual dependence, a ME model generally won't have a unique solution; however, one can be imposed by including an L_2 penalty in the loss term.

This demonstrates that NB and LR are the optimal solvers for simple classification tasks with binary features, depending on what independence assumptions one is willing to make. When features are not binary, the probabilities $P(f_j|c_k)$ and $P(f_j)$ need to be represented with different probability density functions, yielding different gradients and, when possible, solutions. However, in all cases, Bayes' rule must first factorize the posterior into a product of conditionally independent likelihoods and a prior. The ensuing discussion will examine these classifiers from an information-theoretic perspective, and then will show how the score term can be simplified further to a form commonplace in neural machine learning.

3.1.2 Alternative parameterizations

Pointwise Mutual Information

The likelihood and prior terms in Eqs. (3.19, 3.20) both have information-theoretic interpretations. The likelihood is the exponentiated *pointwise mutual information* (PMI) between feature f_j and class c_k, $\text{PMI}(c_k, f_j)$. Pointwise mutual information is a measure of how much more or less frequently two variables occur together than would be expected if they're independent. The prior is the exponentiated negative *self-information* (SI), which measures how surprising it is for an event to occur. The pointwise self-information of a variable x is the same as the PMI of the variable and

itself PMI(x, x), so the whole score from Eq. (3.23) can be expressed solely in terms of PMI.

$$\text{PMI}(x_1, x_2) = \ln\left(\frac{P(x_1, x_2)}{P(x_1)P(x_2)}\right) \tag{3.53}$$

$$s_{ik} = \sum_{j=1}^{d}\left[\ln\left(\frac{P(f_j|c_k)}{P(f_j)}\right)\right] + \ln(P(c_k)) \tag{3.54}$$

$$\ln\left(\frac{P(f_j|c_k)}{P(f_j)}\right) = \ln\left(\frac{P(f_j, c_k)}{P(f_j)P(c_k)}\right) \tag{3.55}$$

$$= \text{PMI}(f_j, c_k) \tag{3.56}$$

$$\ln\left(P(c_k)\right) = -\ln\left(\frac{P(c_k, c_k)}{P(c_k)P(c_k)}\right) \tag{3.57}$$

$$= -\text{PMI}(c_k, c_k) \tag{3.58}$$

$$s_{ik} = \sum_{j=1}^{d}\left[\text{PMI}(f_j, c_k)\right] - \text{PMI}(c_k, c_k) \tag{3.59}$$

Eq. (3.53) presents the definition of PMI, and Eq. (3.54) repeats Eq. (3.23). Eqs. (3.55, 3.57) rewrite the log-likelihood and log-prior in terms of the PMI expression. Eq. (3.59) rewrites the score of class k in example i as the total PMI of the class and features minus the SI of the class. This shows that the score of a class reduces to the total PMI of the class and features minus the baseline PMI of the class. Because the exponential function is monotonically increasing and the normalization term scales each score proportionally, in both NB and ME classifiers, the most probable class is the one that has the most mutual information with the observation and the least information itself. Put another way, both NB and ME models learn a probability for how likely it is to see a class c_k occurring jointly with a feature f_j ($P(f_j, c_k)$), as well as probabilities for how likely it is to see class c_k and feature f_j occurring independently ($P(f_j), P(c_k)$). If c_k and f_j occur together more (or less) often than would be expected by chance, then they deem feature f_j as being predictive (or anti-predictive) of class c_k. In this way, both NB and ME learn simple co-occurrence statistics (in the case of ME, through complex optimization) that they use to classify new observations.

Affine function

The score term in Eq. (3.23; 3.54) is somewhat complex, raising the question of whether it can be reparameterized so that no variable or parameter occurs more than once. In fact, it can be simplified

all the way down to an affine function.

$$s_{ik} = \sum_{j=1}^{d} \left[x_{ij} \ln \left(\frac{p_{kj}^{(fc)}}{p_j^{(f)}} \right) + (1 - x_{ij}) \ln \left(\frac{1 - p_{kj}^{(fc)}}{1 - p_j^{(f)}} \right) \right] + \ln \left(p_k^{(c)} \right) \tag{3.60}$$

$$= \sum_{j=1}^{d} \left[x_{ij} \ln \left(\frac{p_{kj}^{(fc)}}{p_j^{(f)}} \right) - x_{ij} \ln \left(\frac{1 - p_{kj}^{(fc)}}{1 - p_j^{(f)}} \right) + \ln \left(\frac{1 - p_{kj}^{(fc)}}{1 - p_j^{(f)}} \right) \right] + \ln \left(p_k^{(c)} \right) \tag{3.61}$$

$$= x_{ij} \sum_{j=1}^{d} \left[\ln \left(\frac{p_{kj}^{(fc)}}{p_j^{(f)}} \right) - \ln \left(\frac{1 - p_{kj}^{(fc)}}{1 - p_j^{(f)}} \right) \right] + \sum_{j=1}^{d} \left[\ln \left(\frac{1 - p_{kj}^{(fc)}}{1 - p_j^{(f)}} \right) \right] + \ln \left(p_k^{(c)} \right) \tag{3.62}$$

$$w_{kj} = \ln \left(\frac{p_{kj}^{(fc)}}{p_j^{(f)}} \right) - \ln \left(\frac{1 - p_{kj}^{(fc)}}{1 - p_j^{(f)}} \right) \tag{3.63}$$

$$b_k = \sum_{j=1}^{d} \left[\ln \left(\frac{1 - p_{kj}^{(fc)}}{1 - p_j^{(f)}} \right) \right] + \ln \left(p_k^{(c)} \right) \tag{3.64}$$

$$s_{ik} = \sum_{j=1}^{d} \left[x_{ij} \, w_{kj} \right] + b_k \tag{3.65}$$

$$= \mathbf{w}_k^\top \mathbf{x}_i + b_k \tag{3.66}$$

$$\mathbf{s}_i = W \mathbf{x}_i + \mathbf{b} \tag{3.67}$$

Eq. (3.60) is the score from Eq. (3.23), repeated for reference. Eq. (3.61) distributes the $(1 - x_{ij})$ term. Eq. (3.62) factors out the x_{ij} from the two terms inside the summation that include it, and separates out the one term in the sum that doesn't. Eqs. (3.63, 3.64) define new *weight* and *bias* terms using only constants and parameters. The log likelihoods of x_{ij} being 0 are grouped with the log prior in Eq. (3.64) and subtracted from the log likelihoods of x_{ij} being 1 in Eq. (3.63). In this way, when and only when x_{ij} is 1, the copy of $P(f_j = 0|c_k)$ stored in w_{kj} will cancel out with the copy stored in b_k. Eqs. (3.65–3.67) substitute these weights and biases back into the original score, yielding a simple affine function. The affine score in Eq. (3.67) is exactly equivalent to the original score in Eq. (3.60), so it can be substituted in the probability expressions for the two classifiers discussed in the previous section.

$$P^{(\mathrm{NB})}(c_k|\mathbf{f}) = \exp_k(W \mathbf{x}_i + \mathbf{b}) \tag{3.68}$$

$$P^{(\mathrm{ME})}(c_k|\mathbf{f}) = \mathrm{softmax}_k(W \mathbf{x}_i + \mathbf{b}) \tag{3.69}$$

This shows that both NB and ME reduce to an affine function composed with a nonlinear one. In the case of NB, the affine function composes with the exponential function, and in ME, it composes with softmax. Thus NB can be considered an *affine exponential classifier*, and ME an *affine softmax classifier*. Ensuing discussion will focus on the affine softmax classifier, in particular how to extend

it when the probability being modeled is more complex than what was assumed in this section.

This discussion has shown two things: firstly, that the classifier score is essentially learning co-occurrence statistics in the form of PMI; and secondly, that the classifier score can be represented as an affine function. Putting these two together, it can be hypothesized that the affine layers of neural networks are doing something similar. If the affine function is aggregating PMI from the input, then a ReLU layer of a neural network is computing positive PMI, and a tanh layer is approximating a kind of normalized PMI. This admits a new way of conceptualizing exactly what kinds of information neural networks learn.

3.2 Biaffine classification

3.2.1 Fixed-class classification

Section 3.1 shows how the objective probability—$P(c_k = 1|\mathbf{f} = \mathbf{x}_i)$, where c_k is a class variable, $\mathbf{f}$ is a vector of feature variables, and $\mathbf{x}_i$ is a vector of boolean feature values—can be rewritten into one of two forms. If one assumes that the features are both conditionally independent given c_k and mutually independent—the assumptions at play in a Naïve Bayes classifier—then the posterior probability can be expressed in terms of an exponentiated affine function. The optimal weights W and biases $\mathbf{b}$ in this *affine exponential classifier* can be computed analytically.

$$s_{ik} = \sum_{j=1}^{d} \left[\ln \left(\frac{P(f_j|c_k)}{P(f_j)} \right) \right] + \ln(P(c_k)) \tag{3.70}$$

$$= \mathbf{w}_k^\top \mathbf{x}_i + b_k \tag{3.71}$$

$$\mathbf{s}_i = W\mathbf{x}_i + \mathbf{b} \tag{3.72}$$

$$P^{(\mathrm{NB})}(c_k|\mathbf{f} = \mathbf{x}_i) = \exp_k(\mathbf{s}_i) \tag{3.73}$$

Eq. (3.70) repeats the expression for the score in Eq. (3.23), and Eqs. (3.71, 3.72) reparameterize it into an affine function, the process of which is shown in Eqs. (3.60–3.67). Relaxing the mutual independence assumption for a more accurate classifier—as in a Maximum Entropy model—requires explicitly normalizing the probability expression, which can be done with the softmax function. The result is an *affine softmax classifier*. The normalized version doesn't have an analytic optimum, and must be computed through iterative optimization methods.

$$P^{(\mathrm{ME})}(c_k|\mathbf{f} = \mathbf{x}_i) = \mathrm{softmax}_k(\mathbf{s}_i) \tag{3.74}$$

Both classifiers, of course, assume that all features in $\mathbf{x}_i$ are conditionally independent. In some circumstances, this assumption is too strong. In particular, there may be two features f_j and $f_{j'}$ that have one effect on their own but produce a disproportionately stronger or weaker effect when

both are present. The presence of two features can be explicitly expressed as a *feature conjunction* or *conjunctive feature*. This non-additive effect is also known as an *interaction* effect. Conjunctive features can be precomputed and concatenated to the feature vector $\mathbf{x}_i$. Alternatively, the classification score can be extended in order to capture interaction effects explicitly, as in Eq. (3.75).

$$s_{ik} = \sum_{j=1}^{d} \left[\sum_{j' \leq j}^{d} \left[\ln \left(\frac{P(f_j, f_{j'}|c_k)}{P(f_j, f_{j'})} \right) \right] + \ln \left(\frac{P(f_j|c_k)}{P(f_j)} \right) \right] + \ln(P(c_k)) \tag{3.75}$$

The interactive probabilities $P(f_j, f_{j'})$ and $P(f_j, f_{j'}|c_k)$ can be represented with Bernoulli distributions parameterized by $p_{jj'}^{(ff)}$ and $p_{j'kj}^{(ffc)}$, respectively. Then the score can be straightforwardly reparameterized as a biaffine function (cf. Eqs. 3.60–3.67).

$$\sum_{j=1}^{d} \sum_{j' \leq j}^{d} \left[\ln \left(\frac{P(f_j, f_{j'}|c_k)}{P(f_j, f_{j'})} \right) \right] = \sum_{j=1}^{d} \sum_{j' \leq j}^{d} \left[x_{ij} x_{ij'} \ln \left(\frac{p_{j'kj}^{(ffc)}}{p_{j'j}^{(ff)}} \right) + (1 - x_{ij} x_{ij'}) \ln \left(\frac{1 - p_{j'kj}^{(ffc)}}{1 - p_{j'j}^{(ff)}} \right) \right] \tag{3.76}$$

$$u_{j'kj} = \ln \left(\frac{p_{j'kj}^{(ffc)}}{p_{j'j}^{(ff)}} \right) - \ln \left(\frac{1 - p_{j'kj}^{(ffc)}}{1 - p_{j'j}^{(ff)}} \right) \qquad \text{if } j' \leq j \text{ else } 0 \tag{3.77}$$

$$b_k = \sum_{j=1}^{d} \left[\sum_{j' \leq j}^{d} \left[\ln \left(\frac{1 - p_{j'kj}^{(ffc)}}{1 - p_{j'j}^{(ff)}} \right) \right] + \ln \left(\frac{1 - p_{kj}^{(fc)}}{1 - p_{j}^{(f)}} \right) \right] + \ln \left(p_k^{(c)} \right) \tag{3.78}$$

$$s_{ik} = \sum_{j=1}^{d} \left[\sum_{j' \leq j}^{d} [x_{ij'} u_{j'kj} x_{ij}] + w_{kj} x_{ij} \right] + b_k \tag{3.79}$$

$$\mathbf{s}_i = \mathbf{x}_i^\top \mathbf{U} \mathbf{x}_i + W \mathbf{x}_i + \mathbf{b} \tag{3.80}$$

Eq. (3.76) replaces the interaction probabilities with Bernoulli functions. Eqs. (3.77, 3.78) define new parameters in terms of the Bernoulli parameters. Finally, Eqs. (3.79, 3.80) express the score in terms of the new parameters, revealing a simple biaffine function. This biaffine score can then of course be used with the exponential or softmax functions to create a *biaffine classifier*. The second-order term $\mathbf{x}_i^\top \mathbf{U} \mathbf{x}_i$ is known as a *bilinear* transformation, but because first-order terms and constants are added to it, $\mathbf{x}_i^\top \mathbf{U} \mathbf{x}_i + W \mathbf{x}_i + \mathbf{b}$ is a *biaffine* transformation.

It may be the case that there are two *sets* of features $\mathbf{f}$ and $\tilde{\mathbf{f}}$ that are hypothesized to interact with each other but that aren't expected to interact within themselves. The formulation in Eq. (3.80) can easily be extended to accommodate this two-vector variant.

$$P(c_k|\mathbf{f} = \mathbf{x}_i, \tilde{\mathbf{f}} = \tilde{\mathbf{x}}_i) = \text{softmax}_k \left(\mathbf{x}_i^\top \mathbf{U} \tilde{\mathbf{x}}_i + W(\mathbf{x}_i \oplus \tilde{\mathbf{x}}_i) + \mathbf{b} \right) \tag{3.81}$$

Of course, the parameters in Eq. (3.81) are slightly different functions of the Bernoulli parameters

than the ones in Eq. (3.80); for example, each slice of the tensor $\mathbf{U}$ in Eq. (3.80) is triangular, but in Eq. (3.81) each slice is square. Fortunately, the new mapping from Bernoulli parameters to biaffine parameters is straightforward to derive. The probability in Eq. (3.81) includes a bilinear term $\mathbf{x}_i^\top \mathbf{U}\tilde{\mathbf{x}}_i$ to capture interactions between the two feature vectors, as Eq. (3.80). But then it also includes a linear term $W(\mathbf{x}_i \oplus \tilde{\mathbf{x}}_i)$ that captures the individual effects of each feature from each set, plus a bias term $\mathbf{b}$ to capture the prior probability and the likelihood of the class given the absence of each feature or pair of features.

3.2.2 Variable-class classification

The affine and biaffine classifiers so far have all assumed that there is a fixed m number of classes. However, there are cases when the possible classes change from data point to data point. In the prototypical case, the classes are the locations of tokens in a sentence. There, the task is, for a given sentence and input token, to identify the location of exactly one token in the sentence that bears a specific relationship to the input token. In order to classify an input $\mathbf{x}_i$ into one of these location-based classes, each class needs its own feature variables $\tilde{\mathbf{f}}_k$ and input vector $\tilde{\mathbf{x}}_{ik}$, which can be stacked into the matrices $\tilde{F}$ and $\tilde{X}_i$. For example, if an input vector $\mathbf{x}_i$ contains the identity of a token to classify (among other things), each class vector $\tilde{\mathbf{x}}_i$ would similarly contain the identity of the token at a unique location in the sentence. Formally, the goal now is to compute $P(c_k = 1|\mathbf{f} = \mathbf{x}_i, \tilde{F} = \tilde{X}_i)$. Assuming full conditional independence (i.e. no interactions) between $\mathbf{f}$ and the class features $\tilde{F}$ allows the score to be completely factorized (Eq. 3.82). Applying the softmax function then guarantees a valid probability distribution without relying on mutual independence.

$$s_{ik} = \sum_{j=1}^{d} \left[\ln \left(\frac{P(f_j|c_k)}{P(f_j)} \right) \right] + \sum_{j=1}^{d'} \left[\ln \left(\frac{P(\tilde{f}_{kj}|c_k)}{P(\tilde{f}_{kj})} \right) \right] + \ln(P(c_k)) \tag{3.82}$$

$$P(c_k|\mathbf{f}, \tilde{F}) = \text{softmax}_k(\mathbf{s}_i) \tag{3.83}$$

However, this approach—with no interactions between $\mathbf{f}$ and $\tilde{\mathbf{f}}$—is deeply flawed. It's reasonable to assume that all classes are equally likely, in order to accommodate classes not seen at training time (e.g. longer sentences with more positions). With this assumption, it follows that $\ln(P(c_k))$ is

constant for all classes; but constants divide out of the softmax function (as proven below).

$$\text{softmax}_k(\mathbf{z} + c) = \frac{\exp(z_k + c)}{\sum_{k'=1}^{m} [\exp(z_{k'} + c)]} \tag{3.84}$$

$$= \frac{\exp(c)\exp(z_k)}{\sum_{k'=1}^{m} \left[\exp(c)\exp(z_{k'})\right]} \tag{3.85}$$

$$= \frac{\exp(c)\exp(z_k)}{\exp(c)\sum_{k'=1}^{m} [\exp(z_{k'})]} \tag{3.86}$$

$$= \frac{\exp(c)}{\exp(c)} \frac{\exp(z_k)}{\sum_{k'=1}^{m} [\exp(z_{k'})]} \tag{3.87}$$

$$= \frac{\exp(z_k)}{\sum_{k'=1}^{m} [\exp(z_{k'})]} \tag{3.88}$$

$$= \text{softmax}_k(\mathbf{z}) \tag{3.89}$$

Similarly, it's often reasonable to assume that c_k and f_j are mutually independent. In the prototypical case, where classes are locations in a sentence associated with the word at that index, this is akin to saying that an input feature f_j is equally likely no matter the absolute location of its class token; a word related to the token at position 10 is just as likely to have $f_j = 1$ as a word related to the token at position 20. This again accommodates longer sentences, which may have rare or unseen positions. However, it completely nullifies the effect of the likelihood of the input.

$$\ln\left(\frac{P(f_j|c_k)}{P(f_j)}\right) = \ln\left(\frac{P(f_j, c_k)}{P(f_j)P(c_k)}\right) \tag{3.90}$$

$$= \ln\left(\frac{P(f_j)P(c_k)}{P(f_j)P(c_k)}\right) \tag{3.91}$$

$$= 0 \tag{3.92}$$

The end result of this is that the bias $P(c_k)$ and input feature likelihood $P(f_j|c_k)$ disappear from Eq. (3.82), leaving the score conditioned entirely on the likelihood of the class features $P(\tilde{f}_{kj}|c_k)$.

$$s_{ik} = \sum_{j=1}^{d'} \left[\ln\left(\frac{P(\tilde{f}_{kj}|c_k)}{P(\tilde{f}_{kj})}\right)\right] \tag{3.93}$$

This means that by excluding feature interactions, the probability of a class being correct is dependent solely on the features of the class itself, and any features specific to the input are ignored. If there are multiple inputs with the same set of class features, they are all guaranteed to be assigned to the same class, which is rarely if ever desirable behavior.

As foreshadowed in previous discussion, the solution to this problem is to add feature interactions to Eq. (3.82). Any input feature f_j will have the opportunity to interact with any class feature $\tilde{f}_{k\tilde{\jmath}}$. The class bias and likelihood of the input features are still absent, but the input features now affect the score by means of their *joint* likelihood with the class features. This yields the formulation in Eq. (3.94), which can as usual be rewritten into a biaffine function.

$$s_{ik} = \sum_{\tilde{\jmath}=1}^{\tilde{d}} \left[\sum_{j=1}^{d} \left[\ln \left(\frac{P(f_j, \tilde{f}_{k\tilde{\jmath}} | c_k)}{P(f_j, \tilde{f}_{k\tilde{\jmath}})} \right) \right] + \ln \left(\frac{P(\tilde{f}_{k\tilde{\jmath}} | c_k)}{P(\tilde{f}_{k\tilde{\jmath}})} \right) \right] \tag{3.94}$$

$$\mathbf{s}_i = \tilde{X}_i U \mathbf{x}_i + \tilde{X}_i \mathbf{w} \tag{3.95}$$

$$= (\tilde{X}_i U)\mathbf{x}_i + (\tilde{X}_i \mathbf{w}) \tag{3.96}$$

$$= \tilde{X}_i (U\mathbf{x}_i + \mathbf{w}) \tag{3.97}$$

$$P(c_k | \mathbf{f}, \tilde{F}) = \text{softmax}_k(\mathbf{s}_i) \tag{3.98}$$

The variable-class biaffine softmax classifier can be conceptualized in two ways. Firstly, it can be seen as an affine classifier with m_i classes over the input $\mathbf{x}_i$, where the weights and biases are generated on-the-fly for each example (Eq. 3.96). Alternatively, it can be seen as a linear classifier over the class matrix $\tilde{X}_i$, where the vector that weights every $\tilde{\mathbf{x}}_i$ is generated on-the-fly (Eq. 3.97). That is, the scoring function is equal parts affine transformation of the input features and linear transformation of the class features.

3.3 Neural classification

3.3.1 Feedforward networks

One nice property of a feature-based approach is that one can use it to make linguistic observations. If a linguistic feature makes a statistically significant contribution to the classifier, then one can infer (with some caveats) that the feature is in some sense "real". Neural classification sacrifices this transparency for accuracy. Rather than depending on the engineers to come up with useful, predictive features, a neural system aims to learn predictive features automatically. This is done by replacing the high-level features with differentiable, parametric functions that take lower-level information as input. In the domain of natural language, the lower-level information can simply be the set of all words in a sentence. As in the simple affine classifier motivated in Section 3.1, the vector of features $\mathbf{x}_i$ is first put through an affine transformation to rearrange the data without

heavily distorting it. Then, a nonparametric, nonlinear function f is applied to the resulting vector, which allows the system to strip away irrelevant noise.

$$\mathbf{z}_i = \text{Affine}(\mathbf{x}_i) \tag{3.99}$$

$$\mathbf{h}_i = f(\mathbf{z}_i) \tag{3.100}$$

This vector $\mathbf{h}_i$ is known as the *hidden state* or *hidden layer* because it is a latent representation of the input. It is also sometimes referred to as a *fully connected layer* or *feedforward layer* because every node in the hidden state is a linear combination of every element in the input vector. This hidden state can be conceptualized as containing the features that the system learned. It then gets used in the core classifier in the same way that the handcrafted features were used in the preceding sections, with another affine transformation that assigns a high score to the correct class.

$$\mathbf{s}_i = \text{Aff}(\mathbf{h}_i) \tag{3.101}$$

$$P(c_k|\mathbf{f} = \mathbf{h}_i) = \text{softmax}_k(\mathbf{s}_i) \tag{3.102}$$

This composite affine/nonlinearity/affine function is known as a *multilayer perceptron* model, or MLP. It is also sometimes referred to as a *feedforward neural network*, or FFNN. An FFNN model can have multiple hidden layers "stacked" on top of each other, as in Eqs. (3.103–3.105).

$$\mathbf{h}_i^{(1)} = f(\text{Aff}(\mathbf{x}_i)) \tag{3.103}$$

$$\mathbf{h}_i^{(2)} = f(\text{Aff}(\mathbf{h}_i^{(1)})) \tag{3.104}$$

$$\mathbf{h}_i^{(\ell)} = f(\text{Aff}(\mathbf{h}_i^{(l-1)})) \tag{3.105}$$

To simplify notation, this thesis will use the FFNN function (for *feedforward neural network*) defined in Eq. (3.106) to abstract away from the exact number of hidden layers in a neural transformation.

$$\text{FFNN}(\mathbf{x}) = \mathbf{h}^{(L)} \tag{3.106}$$

This FFNN function can then be used to express the score in a neural classifier.

$$\mathbf{s}_i = W\text{FFNN}(\mathbf{x}_i) + \mathbf{b} \tag{3.107}$$

The nonlinearity is necessary for ensuring that the system is more powerful than the simple affine

classifier over handcrafted features. This can be shown in Eqns. (3.108–3.112).

$$s_i = W^{(2)}(W^{(1)}\mathbf{x}_i + \mathbf{b}^{(1)}) + \mathbf{b}^{(2)} \tag{3.108}$$

$$s_i = W^{(2)}W^{(1)}\mathbf{x}_i + W^{(2)}\mathbf{b}^{(1)} + \mathbf{b}^{(2)} \tag{3.109}$$

$$W' = W^{(2)}W^{(1)} \tag{3.110}$$

$$\mathbf{b}' = W^{(2)}\mathbf{b}^{(1)} + \mathbf{b}^{(2)} \tag{3.111}$$

$$s_i = W'\mathbf{x}_i + \mathbf{b}' \tag{3.112}$$

Eq. (3.108) expands the score vector assigned by an MLP with no nonlinearity, representing it as a function of the input vector $\mathbf{x}_i$ and the weight and bias parameters. Eq. (3.109) uses the property of distributivity to distribute the outermost weight matrix $W^{(2)}$ to the two inner terms. Eqns. (3.110, 3.111) substitute the parameter expressions in (3.109) with new single-term parameters. Eq. (3.112) inserts the substitutions back into Eq. (3.109), revealing that the original composite function can be expressed as a single affine transformation. The insertion of a nonlinearity in between the two affine transformations prevents the distribution of the outermost weight matrix in Eq. (3.109), so that a FFNN cannot be reduced to a single affine function. Having motivated the need for a nonlinearity, what should that nonlinarity be? One option is the sigmoid function, which bounds the values of $\mathbf{h}_i$ to be between 0 and 1. Another option, which tends to produce better results, is the hyperbolic tangent function tanh, which bounds $\mathbf{h}_i$ to be between -1 and 1. The most effective popular nonlinear function is known as *ReLU* (rectified linear unit), which keeps all positive values and sets every value less than zero to zero.

Neural classifiers tend to outperform feature-based classifiers empirically, but they wind up being very difficult to interpret. This happens for a number of reasons. Firstly, because the original input is very low-level, figuring out what each latent feature in $\mathbf{h}_i$ represents normally requires examining what parts of each input activate it by looking at the weights and gradients. Secondly, the hidden vectors are continuous, and tend to take advantage of this fact; a single hidden unit will take on a wide range of values across all inputs. Thirdly, the hidden nodes will often represent unintuitive combinations of the input. To illustrate these facts, consider a the matrix X of all

possible combinations of three binary features in Eq. 3.113.

$$X = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{3.113}$$

An *autoencoder* is a neural regression system that aims to convert its input to a hidden state and then reconstruct it. Ignoring bias terms, this would look like Eq. (3.114, 3.115), with the loss defined in Eq. (3.116) (the L_2 loss).

$$H = \tanh(XW^{(1)}) \tag{3.114}$$

$$\hat{X} = HW^{(2)} \tag{3.115}$$

$$\varepsilon = \sum \frac{1}{2}(\hat{X} - X)^2 \tag{3.116}$$

An intuitive solution to Eq. 3.114 would involve letting $W^{(1)}$ be the identity matrix I and setting $W^{(2)}$ to a scalar multiple of the identity matrix αI, which simplifies to Eqs. (3.117–3.119).

$$H = \tanh(X) \tag{3.117}$$

$$X' = \alpha H \tag{3.118}$$

$$\alpha = \frac{1}{\tanh(1)} \tag{3.119}$$

This amounts to purturbing X negligibly when generating the hidden representation, so that H looks almost identical to it and can be easily recovered through simple rescaling. However, the features learned by a neural network can be considerably more complex. In one generated solution,

the hidden representation of each row H has the following form:

$$H = \begin{bmatrix} 0 & 0 & 0 \\ .76 & 0 & .4 \\ 0 & -.42 & 0 \\ 0 & .42 & -.4 \\ .76 & -.42 & .4 \\ .76 & .42 & 0 \\ 0 & 0 & -.4 \\ .76 & 0 & 0 \end{bmatrix} \tag{3.120}$$

In the neural solution, the hidden feature in the first colum $\hat{f}_1$ is positive whenever the input feature f_1 is 1. However, $\hat{f}_2$ is zero when f_3 and f_2 are the same, otherwise negative when f_2 is 1 and positive when f_3 is 1. Similarly, $\hat{f}_3$ is zero when f_3 and f_1 are the same, otherwise negative when f_3 is 1 and positive when f_1 is 1. This is counterintuitive because f_1 is sometimes redundantly represented in the hidden vector, being inferrable when either $\hat{f}_1$ or $\hat{f}_3$ is positive. Additionally, determining whether f_2 is present sometimes requires looking at the whole hidden state: $\hat{f}_2$ is only zero when f_2 and f_3 have the same value, and $\hat{f}_3$ is only zero when f_3 and f_1 have the same value; so when $\hat{f}_2$ and $\hat{f}_3$ are *both* zero, the underlying f_2 and f_3 can only be determined from $\hat{f}_1$, which direcly reflects the value of f_1. Finally, in the intuitive human solution, each node in the hidden state would only ever take on one of two values (including 0); here, nodes $\hat{f}_2$ and $\hat{f}_3$ each take on three. This is hardly the simplest way of solving the autoencoder problem, but without imposing any further constraints, there is no reason for the network to prefer the more human-intuitive solution.

It is worth noting that while the system sometimes learns bizarre features, the solution to this particular toy task always seems to learn *concatenative, real-valued* features, rather than *additive, vector-valued* ones. That is, each column in H has some "meaning" independent of the other columns; $h_{i,2}$ being zero *means* that two particular features in $\mathbf{x}_i$ have the same value. Without the tanh nonlinearity—which is what makes this approach truly "neural", as explained in Eqs. (3.108–3.112), by actively blocking additive representations—a learned solution would only require that $W^{(1)}$ and $W^{(2)}$ be inverses of each other. This means the binary features in F would be represented as whole vectors, and when two features are present, their vector representations would be added together,

as in the solution learned below.

$$H' = \begin{bmatrix} 0 & 0 & 0 \\ -.92 & -.39 & .01 \\ .3 & -.69 & .65 \\ -.25 & .6 & .76 \\ -.62 & -1.08 & .66 \\ -1.17 & .22 & .76 \\ .05 & -.09 & 1.41 \\ -.87 & -.48 & 1.42 \end{bmatrix} \tag{3.121}$$

Here the individual columns of H' do not have meanings that can be clearly related to the original features in X. Instead, each feature gets a vector representation (rows 2–4), and to express in the hidden representation that multiple features are occurring together, the vectors get summed. The non-neural, purely linear approach learns only how to rotate an arbitrary input vector in vector space and then rotate it back. By contrast, even though the neural hidden representation might learn some unusual and unintuitive representation of the data, *it can learn not only discrete propositions, but also how to reason about them.* This ability to learn abstract propositions about the data is one of the characteristics of the neural approach that makes it particularly powerful and promising as a parametric machine learning paradigm.

The point of this discussion is to demonstrate both the strengths and weaknesses of using a neural approach to solve computational tasks. While it is very difficult to figure out what aspects of the input the network is using to make its decision, the network is able to find predictive patterns and details that humans wouldn't think to look for and that can't be efficiently expressed with binary values. This generally results in making them more accurate than feature-based models. Consequently, the position of this thesis is that neural methods should be used as the backbone for any system that aims to achieve peak accuracy. It will be shown, however, that a system that neglects linguistic information will sometimes flounder in the face of one that embraces it.

3.3.2 Recurrent neural networks

One way of representing a sentence or document is as a "bag of words", with binary input features $\mathbf{f}$ representing each word that occurs in a sentence, like in Section 3.1.1. This approach can be improved by using the average of pretrained word embeddings rather than a multi-hot input vector. The bag of embeddings model is still inadequate for many tasks, especially those that require classifying each token in a sentence. The most popular way to incorporate temporal information into a neural network system is to represent each sentence as a sequence of word embeddings that update the hidden state one-at-a-time. This is known as a *recurrent* neural network. The recurrent hidden state $\mathbf{h}_{it}$ is then made to depend on the input $\mathbf{x}_{it}$ (which will normally be a word embedding) as

well as the hidden state of the previous word $\mathbf{h}_{i,t-1}$. $\mathbf{h}_0$—the base case of the recurrence or *initial state*—can either be fixed at 0 or left as trainable parameters (the models reported in the following chapters do the latter). For notational simplicity, the sentence index i will be left out of the equations for the rest of the section.

$$\mathbf{h}_t = f(W\mathbf{x}_t + R\mathbf{h}_{t-1} + \mathbf{b}) \tag{3.122}$$
$$= f(\text{Affine}(\mathbf{x}_t \oplus \mathbf{h}_{t-1})) \tag{3.123}$$

Recurrent layers can be used as input to higher recurrent layers, just as how fully connected layers can be used as input to higher fully connected layers. The ℓth layer of an L-layer network will have the definition in Eq. (3.126).

$$\mathbf{h}_t^{(1)} = f(\text{Affine}(\mathbf{x}_t \oplus \mathbf{h}_{t-1}^{(1)})) \tag{3.124}$$
$$\mathbf{h}_t^{(2)} = f(\text{Affine}(\mathbf{h}_t^{(1)} \oplus \mathbf{h}_{t-1}^{(2)})) \tag{3.125}$$
$$\mathbf{h}_t^{(\ell)} = f(\text{Affine}(\mathbf{h}_t^{(\ell-1)} \oplus \mathbf{h}_{t-1}^{(\ell)})) \tag{3.126}$$

Again, this notation will be simplified by defining a function RNN that takes a sentence as input and returns the topmost recurrent states as output. Permitting sequences of vectors to be equivalently expressed as being "stacked" into matrices admits the notation in Eq. (3.128). Functions that return matrices will be indexed as in Eq. (3.129), with the index before the arguments to the function.

$$X = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T) \tag{3.127}$$
$$\text{RNN}(X) = H^{(L)} \tag{3.128}$$
$$\text{RNN}_t(X) = \mathbf{h}_t^{(L)} \tag{3.129}$$

This recurrent state can then either be used as-is in an affine classifier Eq. (3.130) or it can be first put through a feedforward neural transformation Eq. (3.131).

$$\mathbf{s}_t = \text{Affine}(\text{RNN}_t(X))) \tag{3.130}$$
$$\mathbf{s}_t = \text{Affine}(\text{FNN}(\text{RNN}_t(X))) \tag{3.131}$$

Each recurrent state $\mathbf{h}_t$ will be conditioned on the entire ordered preceding context, allowing it to make predictions based on word order and locality. Of course, this only allows the model to make predictions based on the preceding context, when following words might be just as predictive. The standard way to address this is to concatenate the output of one RNN layer with the output of another RNN layer that saw the sequence of tokens in reverse. This is known as a *bidirectional* RNN

layer, or BiRNN.

$$\overrightarrow{\mathbf{h}}_t = f(\text{Aff}(\mathbf{x}_t \oplus \overrightarrow{\mathbf{h}}_{t-1})) \tag{3.132}$$

$$\overleftarrow{\mathbf{h}}_t = f(\text{Aff}(\mathbf{x}_{T+1-t)} \oplus \overleftarrow{\mathbf{h}}_{t-1})) \tag{3.133}$$

$$\overleftrightarrow{\mathbf{h}}_t = \overrightarrow{\mathbf{h}}_t \oplus \overleftarrow{\mathbf{h}}_t \tag{3.134}$$

At each timestep t, the BiRNN has seen the entire preceding context and the entire following context. This generally makes it more effective than its forward-only counterpart. As before, BiRNN layers can be stacked on top of each other, with the output of a lower BiRNN layer being used as input to the two unidirectional RNNs the higher level. An L-layer deep BiRNN will be represented with the BiRNN function for notational simplicity. BiRNNs can be used in exactly the same way as unidirectional RNNs—as features in a classifier or as input to a FFNN.

3.3.3 Gated recurrent neural networks

Recurrent Neural Networks can, in principle, remember information for an unbounded number of timesteps. In practice, however, they are notoriously unstable. When tanh is used as the nonlinearity, they often run into the *vanishing gradient* problem, whereby the signal coming from inputs far in the past (or future) has been "squashed" by the nonlinearity so many times that it is too weak for the optimizer to pick up on and amplify. With ReLU, the vanishing gradient problem turns into an *exploding gradient* problem, where the recurrent hidden vectors $\mathbf{h}_t$ of very long sequences accumulate extremely large values, causing the optimizer to try to offset them with extremely large weight matrices, which creates even larger values in the recurrent vectors, until they can't be represented with 32-bit floating point numbers. Different initialization schemes have been purported to help on toy problems (Le et al., 2015; Talathi and Vartak, 2015), but currently the most popular approach is to use a more complex, *gated* architecture. There are currently two such widely-used architectures: *Long short-term memory* (LSTM) networks (Hochreiter and Schmidhuber, 1997) and *gated recurrent unit* (GRU) networks (Cho et al., 2014). Long short-term memory networks avoid the vanishing gradient problem by maintaining a separate cell state $\mathbf{c}_t$ that accumulates throughput additively, allowing the state to grow unboundedly large. It avoides the exploding gradient problem, however, because the *output* hidden state $\mathbf{h}_t$ is distinct from this *throughput* hidden state $\mathbf{c}_t$. The output hidden state, which is what other modules of the network will typically use as input, is bounded to have a stable magnitude with the tanh function. The LSTM has been tweaked and modified in many ways since its original conception; Eqs. (3.135–3.141) show the variant assumed in the rest of

this thesis.

$$Z_t = \text{Affine}(\mathbf{x}_t \oplus \mathbf{h}_{t-1}) \tag{3.135}$$

$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{z}_{1,t}) \tag{3.136}$$

$$\mathbf{i}_t = \text{sigmoid}(2 \cdot \mathbf{z}_{2,t}) \tag{3.137}$$

$$\mathbf{f}_t = \text{sigmoid}(2 \cdot \mathbf{z}_{3,t}) \tag{3.138}$$

$$\mathbf{o}_t = \text{sigmoid}(2 \cdot \mathbf{z}_{4,t}) \tag{3.139}$$

$$\mathbf{c}_t = \mathbf{i}_t \odot \tilde{\mathbf{c}}_t + (1 - \mathbf{f}_t) \odot \mathbf{c}_{t-1} \tag{3.140}$$

$$\mathbf{r}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \tag{3.141}$$

Eq. (3.135) computes the activations used elsewhere in the definition of an LSTM as an affine function of the lower input and the output of the LSTM at the previous timestep. Here, Z_t is a $(4 \times d)$ matrix, because the LSTM requires 4 intermediate vectors to compute the $(d \times 1)$-dimensional output vector. The first one Eq. (3.136), $\mathbf{z}_{1,t}$, is bounded with tanh to be between -1 and 1, to avoid saturating the cell state too much. The other three intermediate vectors Eqs. (3.136–3.139) are bounded to be between 0 and 1 with a sigmoid function. The factor of 2 is included here solely for uniformity, to make the sigmoid equivalent to a scaled and shifted tanh: $\text{sigmoid}(2x) = \frac{\tanh(x)+1}{2}$. If excluded, it would simply be absorbed into the learned weight and bias terms. In Eq. (3.140), each element of the cell state at time t is then computed as a linear combination of the corresponding element in the previous cell state $\mathbf{c}_{t-1}$ and the current candidate cell state $\tilde{\mathbf{c}}_t$. The *input gate* $\mathbf{i}_t$ controls which (if any) elements from the candidate cell state get incorporated into the new cell sate, and the *forget gate* $(1 - \mathbf{f}_t)$ determines which (if any) elements from the previous cell state are forgotten. The recurrent output vector is then computed as the bounded value of the cell state $\mathbf{c}_t$, gated again by the *output gate* (Eq. 3.141). The output gate "hides" information in the cell that may be needed later but isn't relevant at the current timestep. LSTM layers can be used exactly like RNN layers, being stacked on top of each other and/or bidirectional. Consequently the LSTM and BiLSTM functions, like the RNN function, can be defined as taking in a sequence of vectors and producing a sequence of vectors as output, while remaining agnostic to the depth of the network.

$$\text{LSTM}(X) = \overrightarrow{H}^{(L)} \tag{3.142}$$

$$\text{BiLSTM}(X) = \overleftrightarrow{H}^{(L)} \tag{3.143}$$

One drawback of the LSTM is that it requires four times as many weights as its "vanilla" RNN counterpart. One way to address this is by *coupling* the input and forget gate; that is, replace $\mathbf{i}_t$ in Eq. (3.140) with $\mathbf{f}_t$. Because f_{tj} and $(1 - f_{tj})$ are guaranteed to sum to 1, the tanh function in Eq. (3.136) is no longer necessary. Following (Greff et al., 2016), the resulting LSTM architecture will be referred to as a *Coupled input-forget long short-term memory* network (CifLSTM). This approach is

conceptually appealing, because it means the cell state and the recurrent state can each be expressed as an interpolation conditioned on exactly one vector.

$$Z_{i,t} = \mathbf{W}^{(input)}\mathbf{x}_{i,t} + \mathbf{W}^{(recur)}\mathbf{r}_{i,t-1} + B \tag{3.144}$$

$$\tilde{\mathbf{c}}_{i,t} = \mathbf{z}_{1,i,t} \tag{3.145}$$

$$\mathbf{f}_{i,t} = \text{sigmoid}(2 \cdot \mathbf{z}_{2,i,t}) \tag{3.146}$$

$$\mathbf{o}_{i,t} = \text{sigmoid}(2 \cdot \mathbf{z}_{3,i,t}) \tag{3.147}$$

$$\mathbf{c}_{i,t} = \mathbf{f}_{i,t} \odot \tilde{\mathbf{c}}_{i,t} + (1 - \mathbf{f}_{i,t}) \odot \mathbf{c}_{i,t-1} \tag{3.148}$$

$$\mathbf{r}_{i,t} = \mathbf{o}_{i,t} \odot \tanh(\mathbf{c}_{i,t}) + (1 - \mathbf{o}_{i,t}) \odot 0 \tag{3.149}$$

$$\tag{3.150}$$

$$\text{CifLSTM}(X) = \overrightarrow{R}^{(L)} \tag{3.151}$$

$$\text{BiCifLSTM}(X) = \overleftrightarrow{R}^{(L)} \tag{3.152}$$

Other tweaks have been recently introduced, such as "highway connections" (Zhang et al., 2016) similar to the simple but effective "residual connections" (He et al., 2016) used in many vision systems, but the work in this thesis only explores the basic LSTM and CifLSTM variants as described above.

One criticism of the LSTM is that it requires computing both a cell state and a recurrent state, which must both be maintained in memory. An alternative recurrent architecture known as a *gated recurrent unit* network (GRU; Cho et al. 2014) aims to avoid this concern by shifting where some of the operations in a CifLSTM occur.

$$\tilde{\mathbf{z}}_t = \text{Aff}(\mathbf{x}_t \oplus \mathbf{h}_{t-1}) \tag{3.153}$$

$$\mathbf{o}_t = \text{sigmoid}(2 \cdot \tilde{\mathbf{z}}_t) \tag{3.154}$$

$$\tilde{\mathbf{h}}_t = \mathbf{o}_t \odot \mathbf{h}_{t-1} \tag{3.155}$$

$$Z_t = \text{Aff}(\mathbf{x}_t \oplus \tilde{\mathbf{h}}_{t-1}) \tag{3.156}$$

$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{z}_{1,t}) \tag{3.157}$$

$$\mathbf{f}_t = \text{sigmoid}(2 \cdot \mathbf{z}_{2,t}) \tag{3.158}$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \tilde{\mathbf{c}}_t + (1 - \mathbf{f}_t) \odot \mathbf{c}_{t-1} \tag{3.159}$$

$$\mathbf{h}_t = \mathbf{c}_t \tag{3.160}$$

The output gate is computed and applied to the previous state at the beginning of the current update (Eq. 3.155), rather than at the end of the previous one (Eq. 3.149). Because the use of a coupled input-forget gate requires only one application of tanh, a GRU network applies it to the candidate

cell (Eq. 3.157) rather than the updated cell (Eq. 3.149). By moving these two components from Eq. (3.149) to a different part of the update, the recurrent state and cell state become equivalent, as in Eq. (3.160). This way, the hidden cell state does not need to be kept independent of the recurrent state. However, some sacrifices are made to achieve this. One is that it requires twice as many matrix multiplications, which hurts most linear algebra libraries' abilities to optimize runtime (on a CPU or GPU). Another is that the network is unable to "hide" nodes that contain information being saved for later. The ramifications of being forced to reveal the entire hidden state will be further examined in Chapter 5.

3.4 Conclusion

This section has derived two different kinds of classifiers based on the standard affine softmax classifier in Eq. (3.161) used for logistic regression and neural classification. The original affine classifier in Eq. (3.161) assumes that the posterior is conditioned on one set of conditionally independent features. The fixed-class biaffine classifier in Eq. (3.162) assumes that the probability of each class is a function of two internally conditionally independent sets of features that are (pairwise) conditionally dependent with each other. The variable-class biaffine classifier in Eq. (3.163) is a sort of combination of the two; in a variable-class scenario, any strictly affine classifier will be insufficient because the input features $\mathbf{f}$ are completely disconnected from the available classes.

$$P(c_k|\mathbf{f} = \mathbf{x}_i) = \text{softmax}_k(W\mathbf{x}_i + \mathbf{b}) \tag{3.161}$$

$$P(c_k|\mathbf{f} = \mathbf{x}_i, \tilde{\mathbf{f}} = \tilde{\mathbf{x}}_i) = \text{softmax}_k(\tilde{\mathbf{x}}_i^\top \mathbf{U}\mathbf{x}_i + W(\tilde{\mathbf{x}}_i \oplus \mathbf{x}_i) + \mathbf{b}) \tag{3.162}$$

$$P(c_k|\mathbf{f} = \mathbf{x}_i, \tilde{F} = \tilde{X}_i) = \text{softmax}_k(\tilde{X}_i U\mathbf{x}_i + \tilde{X}_i\mathbf{w}) \tag{3.163}$$

Utilizing feature interactions reconnects the input features to the class features, allowing for the probability of each class to be conditioned on features of the class as well as features of the input. Additionally, all three classifiers in Eqs. (3.161–3.163) maintain information-theoretic interpretations; the biaffine classifiers differ only in that they include the PMI of the $f_j, \tilde{f}_{k\tilde{j}}$ and c_k. These theoretically-motivated biaffine classifiers will prove to be ubiquitous in the upcoming chapters. Chapter 5 will use a variable-class biaffine classifier and a fixed-class biaffine classifier in tandem to make labeled dependency parsing decisions. It will compare the biaffine approach for classification to the more common feedforward approach, arguing that the biaffine approach is better motivated theoretically and empirically. Chapter 6 will use both affine and biaffine classification in a part-of-speech tagger to condition successively harder tagging decisions on easier ones. It will also use a single-class biaffine classifier and a biaffine regressor to condition parsing decisions on the relative locations of the dependent and the possible head word. Chapter 7 will extend the biaffine parser to a

less common and less restrictive class of dependency formalisms (described at some length in Chapter 2), arguing that the biaffine parser is more principled than a much more complex alternative. Ultimately, much of the remainder of this thesis will demonstrate the utility of biaffine interactions in parsers specifically and neural machine learning more broadly.

Chapter 4

Statistical Parsing

The linguistic theories described in Chapter 2 are only useful for practical applications if their representations can be automatically generated for arbitrary sentences. The goal of this thesis is to build a system that can parse sentences with high accuracy, so this chapter will describe previous and alternative approaches to producing parse trees and lay the foundation for the ensuing system.

4.1 Grammar-based parsing

Syntactic frameworks all assume that at some level, a context-free grammar (CFG) is involved in generating sentences, whether the CFG creates phrase structures or dependency structures. There are efficient ($O(n^3)$) algorithms for taking a manually generated context-free grammar and using it to parse a sentence, such as the CYK algorithm (Cocke and Schwarts, 1970; Younger, 1967; Kasami, 1966) and the Earley algorithm (Earley, 1970). However, two problems arise with using handcrafted grammars to parse. The first issue is that there are many edge cases that handcrafted grammars are likely to miss. If a sentence with a rare construction occurs during inference and no rule exists in the grammar to parse it, then the system is unable to provide even a reasonable guess as to its structure. For example, constructing the parse tree for the sentence in Ex. (4.1) requires unusual rules, like those in Ex. (4.2).

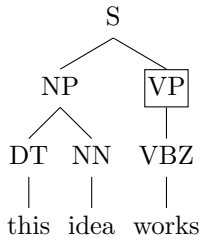
(4.1) The more, the merrier

(4.2) a. S $\rightarrow$ NP NP
b. NP $\rightarrow$ DT JJR

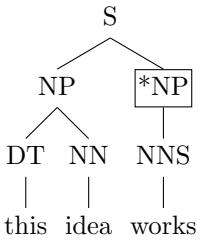
A grammar engineer is unlikely to include the rules in Ex. (4.2) unless they consider this particular construction, meaning the parser is likely to fail on this kind of sentence. If a large annotated treebank is available, then a handcrafted grammar can be augmented with rules inferred from the trees in the treebank, which may include examples of Ex. (4.1). However, expanding the size of

the grammar often introduces spurious ambiguity. For example, adding the rules above creates two possible trees for the simple sentence "this idea works": the correct tree in Ex. (4.3) and the spurious tree that is now possible in Ex. (4.4).

(4.3) This idea works



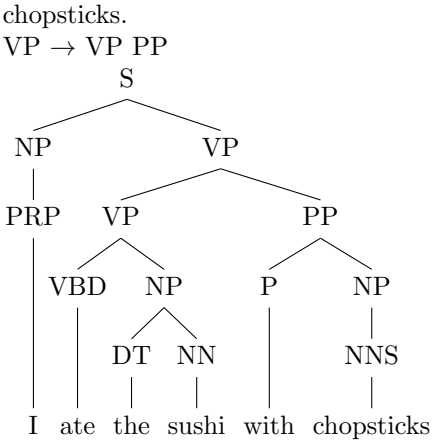
(4.4) This idea works



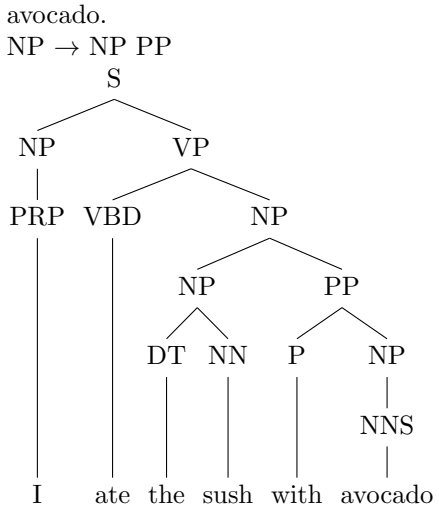
Ambiguity arises naturally in spoken language, but is exacerbated when building context-free grammars using a very small set of syntactic categories. A parser that uses context-free rules will need to either enumerate all possible parses for a given string—which increases more or less exponentially with the number of choice points—or pick one deterministically according to some heuristic. This problem can be alleviated with *probabilistic* context-free grammar rules in a *probabilistic context-free grammar* (PCFG; Booth and Thompson 1973; Baker 1979). Rule probabilities can be computed empirically from the corpus, so that linguistic rules of the form "Y $\rightarrow$ X_1 ... X_n" will have $P(y|x_1, \ldots x_n)$. This conditional probability can be expanded to $\frac{P(x_1,\ldots,x_n,y)}{P(x_1,\ldots,x_n)}$, and then the probabilities in the numerator and denominator can be approximated by simply counting occurrences in a treebank. A parser can then explore all possible parses and return only the one with the highest probability according to the corpus statistics. This comes with its own limitations, however; some expansions observed during training time will never be produced during inference. For example, consider the prepositional attachment ambiguity in Ex. (4.5a, 4.5b).

(4.5) I ate the sushi with...

a. chopsticks.



b. avocado.



Given the sequence VBD NP PP, there are two possible trees that can be constructed: one where the PP modifies the verb Ex. (4.5a), and another where it modifies the noun Ex. (4.5b). The actual lexical items are what determine which structure is correct—it would be extremely surprising to discover that someone used avocado as a utensil, or to hear about sushi adorned with chopsticks. However, the probabilities assigned to the rules that dictate the sentences' structures are not conditioned on the lexical items; the words themselves play no part in the decision to use one rule over the other. Consequently, one of the two structures is guaranteed to maximize the probability of both sentences, so that one sentence is guaranteed to be wrong at inference time.

The problem of overly coarse syntactic categories can be helped by annotating categories based on features of the surrounding environment (Collins, 2003; Klein and Manning, 2003). For example, they can include the category immediately higher in the tree, to distinguish between subjects (NPˆS) and objects (NPˆVP). This is also known as *vertical markovization*.

(4.6) SˆTOP → NPˆS VPˆS

(4.7) VPˆS → VBDˆVP NPˆVP

This is equivalent to predicting the *grandparent* node given a sequence of categories and their parent. Categories can also be annotated for morphological features, such as *singular* and *plural*. This will help prevent the system from predicting trees that violate subject-verb agreement rules.

(4.8) S-SG → NP-SG VP-SG

(4.9) S-PL → NP-PL VP-PL

The problem raised in Ex. (4.5) can be solved by annotating each phrase with the lexical item of its functional head. This gives the parser access to the lexical information needed to disambiguate the two possible structures for a VBD NP PP sequence.

(4.10) VP(eats) → NP(sushi) PP(chopsticks)

(4.11) NP(sushi) → NP(sushi) PP(avocado)

When some combination of annotations are insufficient, all possible parses for a sentence can be enumerated and re-ranked by an additional machine learning classifier that uses a different (potentially richer) set of features. Of course, these annotation approaches solve the problem of ambiguity by introducing the problem of sparsity. These three annotations expand the number of syntactic categories by a factor of $2 \times m \times |V|$, and the number of possible rules by $b^{2m|V|}$, where b is the average branching factor. The canonical PCFG model doesn't make any independence assumptions among the annotated features, meaning that it can't produce a category it hasn't seen before or take advantage of the effects of each annotated feature independently. That is, the model doesn't learn any kind of relationship between the sequence "VBZˆVP-SG NPˆVP-SG"—a singular verb and a singular object—and the sequence "VBZˆVP-SG NPˆVP-PL"—a singular verb and a plural object.

As far as the model's concerned, the righthand side of the rule is one large feature conjunction (where the categories also count as features) with no inherent internal structure and no assumption of conditional independence. One might observe that these features have complex interactions that the PCFG approach is poorly equipped to handle. For example, some verbs require direct objects (at least in unmarked constructions), whereas others permit them optionally and still others disallow them entirely; a noun following an intransitive verb cannot be its direct object, and must be something else (like a temporal modifier, which would attach higher). Moreover, agreement in various forms is common across languages, and is most sensibly handled with some sort of *unification* principles that can correctly handle underspecificity. Finally, lexical dependencies such as in Ex. (4.5) arise from semantic and pragmatic attributes of words. One alternative to PCFGs is to use a more structured syntactic representation than trees of discrete symbols, such as HPSG-style attribute-value matrices (Pollard and Sag, 1994), that emphasizes unification constraints rather than rules. However, the increased complexity of the richer representation comes with its own costs—producing a single disamibiguated HPSG parse for a sentence is significantly more involved than producing a simple PCFG parse (Toutanova and Manning, 2002).

While the unification-based approach underspecifies the syntactic categories and requires fully-specified syntactic rules, the solution that has gained the most traction recently can be thought of as learning underspecified or *partial* rules of fully-specified categories. That is, rather than learning the kind of fully-specified rule in Ex. (4.12), these systems learn the underspecified one in Ex. (4.13).

(4.12) VP $\rightarrow$ VBZ NP

(4.13) VP $\rightarrow$ ... NP ...

While the first rule is complete and details the entire expansion of the category VP, the second rule only describes one word or phrase in the expansion. The systems aim to return the tree that maximizes the joint probability of all its partial subtrees. These learned underspecified rules can still for the most part take advantage of the features annotated in the conventional PCFG. Unlike the PCFG though, the systems consider the effect of each feature on the probability of the rule *independently*, rather than (or in addition to) only looking at the effect of multiple features *jointly*.

4.2 Transition-based Parsing

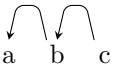
4.2.1 The shift-reduce algorithm

Transition-based dependency parsing algorithms adapt a traditional constituency parser known as a *shift-reduce parser* (Aho and Ullman, 1972), and can be used to parse with handcrafted rules or statistical machine learning techniques (Nivre et al., 2006, 2007). The most well-known variants of the transition-based algorithm are those of Nivre (2004), and are known as *arc-standard* and *arc-eager*. In both variants, a queue called the *input buffer* and a stack (simply called *the stack*)

are initialized. First, every word in the sentence is added to the queue in order. The algorithm then iteratively chooses from a small set of actions to manipulate the state of the queue and buffer, depending on the handcrafted grammar or features of the surrounding context. Arc-standard (which is similar to the algorithm of Yamada and Matsumoto (2003)) is the simpler of the two algorithms and contains only three possible actions: move a word from the buffer to the stack (*shift*); pop the two words from the top of the stack, assign an arc from the first to the second, and push the first back onto the stack (*left-reduce*); pop the two words from the top of the stack, assign an arc from the second to the first, and push the second back onto the stack (*right-reduce*). The arc-standard algorithm has an upper runtime bound of $2n$ operations—where n is the length of the sentence—and is guaranteed to produce a tree that is acyclic and projective (i.e. there are no crossing dependency edges), though potentially unconnected (i.e. there may be words that haven't been assigned a head). This very efficient bound requires that the parser only produce a single tree, rather than enumerating all possible trees that can be generated by the grammar. In a machine-learning-based parser, the single best transition is chosen at each iteration, and in a rule-based parser, the transitions are assigned different priorities and the highest-priority transition possible for each iteration is chosen (Nivre 2003; so for example, the system can be made to left-reducing whenever possible, otherwise to right-reduce if possible, and to only shift as a last resort). Note that assigning an arc from one word to another is equivalent to predicting *part* of a context-free syntax rule, which allows the system to predict syntactic rules it has never seen before. What's more, the features usable in statistical systems overlap with the annotations used in a PCFG—siblings, grandparents, or grandchildren of partially constructed parse trees, extracted morphological features, lexical items, nearby neighbors, part-of-speech tags, etc, but can also easily extend to real-valued neural features. Additionally, these can be conjoined into conjunctive features (cf. Chapter 3.2.1), but do not have to be.

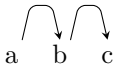
Nivre (2004) notes that the arc-standard algorithm parses bottom-up, such that once a dependent has been attached to a head, it cannot receive any more dependents. The consequence of this is that parsing is not done *incrementally*, where words attach to their heads as soon as both the word and the head are salient on the stack and buffer. To illustrate, in a phrase where every word depends on the word immediately to its right (Figure 4.1a, with arrows pointing *from* heads and *to* dependents), the bottom-up strategy would create each arc as soon as possible. However, in the reverse case (Figure 4.1b), it would have to keep postponing assigning edges until all inputs have been pushed onto the stack. This means that the arc-standard algorithm is implausible as a psychological parsing model (Marslen-Wilson, 1973; Frazier, 1987).

Nivre (2003, 2004) addresses this by adapting the *arc-eager* algorithm originally developed by Abney and Johnson (1991) for constituency parsing to dependency parsing. This algorithm makes leftward attachment decisions bottom-up and rightward attachment decisions top-down. Building this into a dependency parser involves simply redefining the set of actions available: pop the first word from the stack and assign an edge to it from the first word on the buffer (*left-arc*); assign an



action	stack	buffer	edges
	$[]$	$[a, b, c]$	$\{\}$
shift	$[a]$	$[b, c]$	$\{\}$
shift	$[a, b]$	$[c]$	$\{\}$
L-reduce	$[b]$	$[c]$	$\{a \leftarrow b\}$
shift	$[b, c]$	$[]$	$\{a \leftarrow b\}$
L-reduce	$[c]$	$[]$	$\left\{\begin{matrix} a \leftarrow b \\ b \leftarrow c \end{matrix}\right\}$

(a) Parse of a left-branching tree.

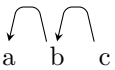


action	stack	buffer	edges
	$[]$	$[a, b, c]$	$\{\}$
shift	$[a]$	$[b, c]$	$\{\}$
shift	$[a, b]$	$[c]$	$\{\}$
shift	$[a, b, c]$	$[]$	$\{\}$
R-reduce	$[a, b]$	$[]$	$\{b \rightarrow c\}$
R-reduce	$[a]$	$[]$	$\left\{\begin{matrix} a \rightarrow b \\ b \rightarrow c \end{matrix}\right\}$

(b) Parse of a right-branching tree.

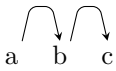
Figure 4.1: An arc-standard transition sequence.

edge to the first word on the buffer from the first word on the stack, and shift it onto the stack (*left-arc*); pop the stack (*reduce*); shift the first word from the buffer to the stack (*shift*). The arc-eager transition system assigns leftward edges to words that have already found all their dependents, but assigns rightward edges to words without blocking them from taking on their own dependents. This is shown in Figure 4.2.



action	stack	buffer	edges
	$[]$	$[a, b, c]$	$\{\}$
shift	$[a]$	$[b, c]$	$\{\}$
left-arc	$[b]$	$[c]$	$\{a \leftarrow b\}$
left-arc	$[c]$	$[]$	$\left\{\begin{matrix} a \leftarrow b \\ b \leftarrow c \end{matrix}\right\}$

(a) A parse of a left-branching tree



action	stack	buffer	edges
	$[]$	$[a, b, c]$	$\{\}$
shift	$[a]$	$[b, c]$	$\{\}$
right-arc	$[a, b]$	$[c]$	$\{a \rightarrow b\}$
right-arc	$[a, b, c]$	$[]$	$\left\{\begin{matrix} a \rightarrow b \\ b \rightarrow c \end{matrix}\right\}$
reduce	$[a, b]$	$[]$	$\left\{\begin{matrix} a \rightarrow b \\ b \rightarrow c \end{matrix}\right\}$
reduce	$[a]$	$[]$	$\left\{\begin{matrix} a \rightarrow b \\ b \rightarrow c \end{matrix}\right\}$

(b) A parse of a right-branching tree

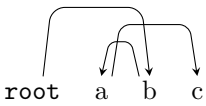
Figure 4.2: An arc-eager transition sequence.

Both the arc-standard and arc-eager shift-reduce parsers guarantee an extremely efficient worst-case runtime of $2n$ transitions. However, because they are adapted from the constituency parsing literature, they also require that the dependency trees behave in certain critical ways like constituency trees. Constituency trees guarantee that each word has at most one head, and as a result these shift-reduce parsers cannot produce multi-headed dependents. The most popular dependency formalism variants (e.g. uncollapsed SD and basic UD) have historically disallowed multi-headed dependents, so the latter has not been a problem that needed to be addressed until recently (cf. the

SemEval 2014 shared task; Oepen et al. 2014). Constituency trees are also by definition projective, with no crossing edges, and as a result these systems are likewise limited to producing projective trees. One of the principal advantages of dependency representations over constituency representations is their ability to permit crossing dependencies, so the fact that the transition-based parsers described above are unable to construct non-projective trees is actually a severe drawback.

Covington (2001) proposes a non-projective transtion-based algorithm that uses two stacks rather than just one: one *word* stack keeps track of all words that have been processed so far, and one *head* stack keeps track of all words that haven't been assigned heads. When a word is popped from the buffer, the algorithm first scans through all words in the *head* stack—which currently lack dependents—and attaches any that the grammar or statistical classifier permits. Then it scans through the words on the *word* stack—everything popped from the buffer—and tries to assign the current word as one of their dependents. If it fails this last step, then it adds it to the dependents stack. The ability to predict non-projective dependencies comes at a cost though, as the $O(n)$ bound that the projective algorithms guarantee is lost and replaced with a $O(n^2)$ bound. Attardi (2006) extends Yamada and Matsumoto's (2003) variation of arc-standard by adding more possible parser actions. Firstly, he allows the parser to "skip over" up to a fixed number of words on the stack and buffer when assigning arcs. For instance, an edge can be assigned between the first word on the buffer and any of the first, second, or third word on the stack, not just the first one. Adding these transitions maintains the linear complexity, but is insufficient to generate arbitrary non-projective trees. In another version, he employs a second, "temporary" stack, where words can be moved to and from (by *extract* and *insert* transitions). This effectively "rearranges" the words in the sentence into an order that can be parsed projectively. The temporary stack allows arbitrary non-projective trees, but results in quadratic worst-case complexity, and requires an additional data structure absent from the basic shift-reduce algorithm. Nivre (2009) instead adds the *swap* transition to the arc-standard algorithm, which turns out to be sufficient for producing arbitrary non-projective trees without adding any more data structures. The swap transition is similar to the insert/extract transitions of Attardi (2006), but without the use of an additional data structure. It simply moves the second word immediately under the topmost word of the stack back onto the buffer, demonstrated with the arc-standard system in Figure 4.3. This elegant solution requires no additional data structures and maintains the $O(n)$ best-case runtime, but again at the cost of a $O(n^2)$ worst-case runtime. Since human language tends to be most naturally organized into mostly projective trees, with crossing arcs being fairly uncommon for most languages, the swap transition is empirically rarely utilized to its full n^2 potential.

In these transition systems, transition sequences are very fragile and susceptible to error propagation, with one wrong transition early on potentially derailing the entire parse and producing a tree riddled with seemingly senseless mistakes. This problem is exacerbated by the fact that transition systems can only manipulate one of two tokens at a time. Long sentences can have very long

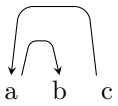


action	**stack**	**buffer**	**edges**
	$[]$	$[\texttt{root}, a, b, c]$	$\{\}$
shift	$[\texttt{root}]$	$[a, b, c]$	$\{\}$
shift	$[\texttt{root}, a]$	$[b, c]$	$\{\}$
shift	$[\texttt{root}, a, b]$	$[c]$	$\{\}$
swap	$[\texttt{root}, b]$	$[a, c]$	$\{\}$
shift	$[\texttt{root}, b, a]$	$[c]$	$\{\}$
shift	$[\texttt{root}, b, a, c]$	$[]$	$\{\}$
R-reduce	$[\texttt{root}, b, a]$	$[]$	$\{a \rightarrow c\}$
L-reduce	$[\texttt{root}, b]$	$[]$	$\{a \rightarrow c, a \leftarrow b\}$
R-reduce	$[\texttt{root}]$	$[]$	$\{a \rightarrow c, a \leftarrow b, \texttt{root} \rightarrow b\}$

Figure 4.3: An arc-standard transition sequence with the swap transition.

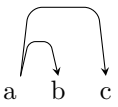
transition sequences, where any decision could depend on information that can only be ascertained from looking very far forward into the buffer or backwards on the stack, increasing the probability of making an error that gets propagated. In both arc-standard and arc-eager, it's easy for the system to remove words from the stack too soon because the system doesn't have access to later tokens that relate to them. Ideally, in a projective dependency formalism, the system should only prevent a word from taking any more dependents when it assigns an edge between two words on either side of it, blocking the word from taking any more dependents that don't involve crossing branches. Qi and Manning (2017) modify the arc-eager transition system to successfully accomplish this. In their transition system, which they dub *arc-swift*, the system can assign an edge between *any* word on the stack and the word at the front of the buffer. All words in between can be safely reduced because they can't take any more dependents from the buffer without crossing the newly-created edge. This both decreases the degree of long-term planning that needs to be made in order to parse a sentence and decreases the length of transition sequences, at the cost of increasing the number of actions possible on each step. The theoretical runtime now increases to $O(n^2)$ as well because the word on the buffer needs to examine all words on the stack when deciding on an action, though as with swapping transition systems this upper bound is rarely reached in practice. This gives it the best of both worlds, and then some—like arc-eager, it parses left-to-right rather than bottom-up, and like arc-standard, it doesn't require a separate reduce transition to close off words. The arc-swift system is shown in Figure 4.4. Note that at any point, all words on the stack stack are capable of taking dependents that won't cross any edges already assigned. The authors find that parsers trained using the arc-swift transition system significantly outperform identical systems with different

transition systems. They find that this improvement is driven by longer dependency edges, presumably because the alternatives run the risk of prematurely reducing the head from the stack, rather than leaving it available to accept dependents for as long as possible. The authors also find that the average transition sequence length under an arc-swift transition system is about three-quarters as long as of arc-eager, meaning that there are fewer decision points where the system could make errors, although it should be noted that each decision point will likely have more actions to choose from.



action	**stack**	**buffer**	**edges**
	$[]$	$[a, b, c]$	$\{\}$
shift	$[a]$	$[b, c]$	$\{\}$
R-arc[1]	$[a, b]$	$[c]$	$\{a \rightarrow b\}$
L-arc[2]	$[]$	$[c]$	$\begin{Bmatrix} a \rightarrow b \\ a \leftarrow c \end{Bmatrix}$
shift	$[c]$	$[]$	$\begin{Bmatrix} a \rightarrow b \\ a \leftarrow c \end{Bmatrix}$

(a) A transition sequence using L-arc[2].



action	**stack**	**buffer**	**edges**
	$[]$	$[a, b, c]$	$\{\}$
shift	$[a]$	$[b, c]$	$\{\}$
R-arc[1]	$[a, b]$	$[c]$	$\{a \rightarrow b\}$
R-arc[2]	$[a, c]$	$[]$	$\begin{Bmatrix} a \rightarrow b \\ a \rightarrow c \end{Bmatrix}$

(b) A transition sequence using R-arc[2].

Figure 4.4: An arc-swift transition sequence.

4.2.2 Neural transition-based models

Chen & Manning 2014

Chen and Manning (2014) built the first dependency parser using neural machine learning techniques that achieved strong performance. They point out that feature-based approaches' reliance on huge feature sets creates a huge tension between speed and accuracy. They further demonstrate that multiplicative interactions between pairs of features—which expands the feature space even further—is necessary for achieving high performance. These are effectively conjunctive features, making the approach with multiplicative interactions equivalent to a biaffine classifier, discussed in Chapter 3.2. However, these complex features require time to generate and look up when parsing, and each new feature added to the feature space increases memory and runtime overhead. Rather than using handcrafted features to identify relevant information about the stack and buffer to a classifier, they instead use vector-valued *word*, *part-of-speech*, and *label* embeddings. In their system, these embeddings come from the following sources:

- First three words at the top of the buffer
- First three words at the top of the stack

- For the two words at the top of the stack, their leftmost children
- For each of them, *their* two leftmost children
- For the two words at the top of the stack, their two rightmost children
- For each of them, *their* two rightmost child
- For each word, its corresponding POS tag
- For each word already assigned a head, its corresponding label

This amounts to 18 word embeddings, 18 tag embeddings, and 12 label embeddings, for 48 feature embeddings in total. Each embedding is a 50-dimensional vector, yielding an input representation of $48 \times 50 = 2,400$ real values. They report that in their experiments, the feature-based approaches generate around $1,000$ times more features than the neural network. By using vector space embeddings rather than feature templates, they vastly reduce the number of features that must be extracted and weighted in order to parse a sentence. The concatenated embeddings are then used in a feedforward neural network (FFNN) with one hidden layer. The output is the vector of transition scores, with the highest scoring transition being the one chosen. Letting $\mathbf{x}_t^{(\text{word})}$ be the concatenated relevant word embeddings for a sentence at transition t, and similar for $\mathbf{x}_t^{(\text{tag})}$ and $\mathbf{x}_t^{(\text{label})}$:

$$\mathbf{x}_t = \mathbf{x}_t^{(\text{word})} \oplus \mathbf{x}_t^{(\text{tag})} \oplus \mathbf{x}_t^{(\text{label})} \tag{4.1}$$

$$\mathbf{h}_t = \text{FFNN}(\mathbf{x}_t; f = \lambda x : x^3) \tag{4.2}$$

$$\mathbf{s}_t = \text{Affine}(\mathbf{h}_t) \tag{4.3}$$

Normally it's very rare for feedforward networks to use a nonlinearity other than tanh or ReLU; however, Chen and Manning (2014) want to build into their model the ability to capture the kind of multiplicative interactions that they found to be useful in feature-based models. One way to do this for feature triplets would have been to use a triaffine layer, with a fourth-order tensor ${}^4\mathbf{U} \in \mathbb{R}^{(d' \times d \times d \times d)}$ (where d' is the size of the hidden layer and d is the size of the input layer):

$$\mathbf{z}_t = (((({}^4\mathbf{U}\mathbf{x}_t)^{\top(4,3)}\mathbf{x}_t)^{\top(4,2)}\mathbf{x}_t + \dots \tag{4.4}$$

$$z_{tk} = \sum_{j=1}^{m}\sum_{j'=1}^{m}\sum_{j''=1}^{m}[w_{kjj'j''}(x_{tj}x_{tj'}x_{tj''}) + \dots] \tag{4.5}$$

Of course, such a system would have $2,400^3 = 13.8$B parameters *for each hidden unit*. With 200 hidden units, that amounts to nearly 2.76T parameters, which needless to say would work counter to their goal of improving speed by trimming parameters. In order to include multiplicative interactions without exploding the parameter space (a goal that the work of this thesis shares), they use the

cube function to implicitly implement a decomposed triaffine transformation.

$$\mathbf{z}_t = W\mathbf{x}_t \tag{4.6}$$

$$z_{tk} = \sum_{j=1}^{m} [w_{kj}x_{tj}] \tag{4.7}$$

$$h_{tk} = \left(\sum_{j=1}^{m} [w_{kj}x_{tj}]\right)^3 \tag{4.8}$$

$$= \sum_{k=1}^{m}\sum_{k'=1}^{m}\sum_{k''=1}^{m} (w_{jk}w_{jk'}w_{jk''})(x_{tk}x_{tk'}x_{tk''}) + \dots \tag{4.9}$$

Rather than represent each weight $w_{jkk'k''}$ individually—which could cost trillions of parameters—they represent it as being multiplicatively composed of three other weights $w_{jk}w_{jk'}w_{jk''}$, reducing the number of parameters by more than six and a half orders of magnitude. This is similar to the work of Lei et al. (2014), who *explicitly* implement a decomposed triaffine transformation in a non-neural system to reduce the parameter space while maintaining complex feature interactions. Chen and Manning (2014) find fairly strong evidence that this approach works better than tanh, but did not compare it to ReLU, which has superseded tanh as the nonlinearity of choice for feedforward networks. However, it might be noted that the cube nonlinearity is being used to convert an affine transformation into a decomposed triaffine one, not to strip away superficial information in the data like tanh or ReLU do. Because the cube function and ReLU are designed for different purposes, there's no reason they shouldn't be combined in feedforward networks where features are expected to have complex interactions. Chen and Manning (2014) found that indeed their parser was not only faster than the feature-based alternatives, but more accurate as well.

Feedforward models

The main contribution of Chen and Manning (2014) was to produce a simple neural dependency parser that achieved very good performance for its time while simultaneously speeding up inference. The speed, accuracy, and simplicity of Chen and Manning's (2014) transition-based parser has led to quite a few extensions of it. Weiss et al. (2015), for instance, use it as a starting point for their work. They start by tuning the hyperparameters more carefully: they use a deeper (two-layer) network, substitute the cubic nonlinearity with ReLU, use smaller label/POS vectors, and use a more involved optimization scheme (SGD with momentum (Hinton, 2012), L_2 regularization, and temporal averaging (Moulines and Bach, 2011)) compared to AdaGrad (Duchi et al., 2011). Additionally, they augment the original parser with beam search to allow it to make globally optimal but locally suboptimal transitions. After (pre-)training the parser without beam search, they freeze the weights for its hidden representations and retrain the linear classifier that makes predictions, this time *with* beam search. The authors find that keeping these pre-training and beam-training

phases distinct improved performance. Neural networks are well-known to flourish in an abundance of data, but dependency treebanks are expensive to annotate, so they additionally utilize *tri-training*. Tri-training is an unsupervised method where two different fully-trained parsers parse an unlabeled corpus, and sentences for which they produce the same parse are added to the training set (Li et al., 2014). This likewise gives Weiss et al. (2015) substantial gains in accuracy.

Alberti et al. (2015) extend Weiss et al.'s (2015) work further. They add a few new innovations. Firstly, they provide the system with morphological features. The morphological features for token t are assigned trainable embeddings and averaged together before being concatenated to the other embeddings used as input to the FFNN. Secondly, rather than only using the most probable tag for token t as input to the system, they weight the top k tags (where k is a tunable parameter) by how much probability the tagger assigned to each of them. These are then looked up in an embedding matrix and summed together. Finally, they use a technique known as *integrated tagging and parsing* to simplify the pipeline. In an integrated system, the transition system is given the ability to classify each token's part-of-speech tag when shifting it onto the stack and before making any edge assignments with it. They find that each of these additions improves performance over Weiss et al.'s (2015) baseline on the CoNLL 09 shared task dataset (Hajič et al., 2009), with the conjunction of all of them generally achieveing the highest performance.

Andor et al. (2016) note that Chen and Manning (2014), Weiss et al. (2015), and Alberti et al. (2015) all use models trained with *local* objectives. That is, at timestep t, the system is trained to maximize the probability of the correct transition with respect to the input $\mathbf{x}_t$ alone. The probability of a whole transition sequence is simply the product of the probabilities of each transition independently. This inhibits the system's ability to learn features that put it in a good position to make more accurate predictions at later timesteps. Thus they extend the system further by training it with a *global* objective, known as a *conditional random field* (CRF). In their system, they optimize not individual decisions, but whole *chains* of decisions. This way, mistakes that completely derail the parse are penalized much more harshly than one-off mistakes that only affect leaf nodes. Unfortunately, optimizing a CRF objective exactly in this context is intractible because it involves summing over all possible transition sequences, of which there are m^T. Andor et al. take advantage of beam search to sample the most probable wrong transition sequences, and update the model parameters when the correct sequence falls off the beam (and/or at the end of the transition sequence). They apply this strategy to both tagging and parsing as a faster alternative to recurrent networks, finding that using this CRF objective improves the performance of Alberti et al.'s (2015) parser.

LSTM models

Chen and Manning (2014) et seq. committed to using feedforward networks instead of recurrent neural networks because of their relative speed and simplicity. However, they're limited in the

amount of context they can take advantage of when making a decision, even when using a CRF objective. Kiperwasser and Goldberg (2016) construct a transition-based model similar to Chen and Manning's, with a feedforward layer that takes as input embeddings of words in key locations on the stack and buffer. However, rather than using embeddings looked up from a dictionary directly, they run a kind of recurrent neural network known as a *bidirectional long short-term memory network* (BiLSTM; (Graves and Schmidhuber, 2005)) over the sequence of token embeddings first. This gives each token a new, contextualized BiLSTM representation, which is given to the feedforward transition classifier instead of the simple word vector. As a result, the system always has access to the entire sequence when making transition decisions, which mitigates the problems that arise from making local decisions without sufficient lookahead. However, transition decisions are still conditioned on the representations of words at manually-specified locations on the stack and buffer; determining which locations are optimal is thus a matter of trial and error.

Dyer et al. (2015) take a different approach, using *unidirectional* LSTMs to alleviate this limitation. While Chen and Manning (2014) represent the stack, buffer, and transition sequence as simple word/tag/label embeddings, Dyer et al. (2015) represent them with LSTMs over word and tag embeddings. This means that each word in the buffer (where the sentence is reversed before being fed into the LSTM) has access to information about what words are coming up later, allowing the system to learn how to avoid reducing words too soon. Like with Kiperwasser and Goldberg, the t-th LSTM state of the stack will have information about all candidate head words up to that point, making it easier for the system to tell whether the current word's head is somewhere on the stack even if its head is very far back. Dyer et al.'s system—which they call the *stack-LSTM*—includes a recursive feedforward composition function as well, incorporating information from dependents into the head's stack representation, allowing the system to avoid some incoherent parses (such as assigning two subjects to one verb). Similarly, the whole transition sequence (including labels) is provided via LSTM, so the system can learn which transition sequences or subsequences are likely to come next. They report substantial improvements with this setup over Chen and Manning's system, especially on the non-western benchmark (Chinese).

Ballesteros et al. (2015) aim to improve the robustness of the stack-LSTM system to different languages. They add the swap transition to allow for non-projective trees—necessary for languages with more free word order—and they build in a character-level word embedding. Their character-level word embedding is composed using a bidirectional LSTM that takes as input character embeddings. The last output vectors of the forward and backward states are then concatenated and used in place of the token embedding. They find that using the character-level embedding consistently allows the system to compensate when the part-of-speech tag is absent; however, when the part-of-speech tag is included, whole-token and character-level systems perform about on par with each other. They do not compare against systems with all three representations (whole-token, character-level, and part-of-speech).

Ballesteros et al. (2016) add to the system a *dynamic oracle* (Goldberg and Nivre, 2012, 2013). The systems described in the previous discussion were all trained with static oracles, which define one correct sequence of transitions. That is, static oracles don't distinguish between transition sequences that produce trees with only one mistake and sequences that produce nonsense parses. This means it's impossible to train a system to learn how to recover from mistakes using a static oracle. Dynamic oracles, by contrast, define the best transition sequence from each possible transition state. If a parser is allowed to make a mistake during training that takes it to a state from which the gold parse tree is impossible to construct, a dynamic oracle can still guide it to the sequence that creates the tree with the fewest errors. Ballesteros et al. (2016) take advantage of dynamic oracles by testing two possible ways they can be used. Firstly, they consider sampling randomly from the distribution of actions predicted by the parser. If the parser assigns a 25% probability to action a and a 75% probability to action b, then there will be a 25% probability that the parser takes action a and a 75% chance it takes action b. They also consider an approach that smoothes the probability distribution with a tunable hyperparameter, so that the system is more likely to take an action other than the one it identifies as most probable. This latter strategy encourages exploration beyond what the model would naturally do, which is ideal because it will likely start to fit the training set as training proceeds. That is, the parser is expected to make more mistakes during inference than during the end stages of training when it has learned the ins and outs of the training set, and reweighting the probability distribution helps to offset this. The authors find that while switching from the arc-standard transition system to the arc-hybrid transition system (which has a more convenient dynamic oracle) hurts performance when using a static oracle, using a dynamic oracle more than makes up for this loss in accuracy. Smoothing the probability distribution to encourage more exploration pushes the accuracy up even more substantially.

Dyer et al. (2016) modify the system from Dyer et al. (2015) in several key ways, calling the new system the *recurrent neural network grammar* (RNNG). Firstly, they change the composition function to use a BiLSTM instead of a recursive FFNN (but use the system as a constituency parser and a language model rather than as a dependency parser). More critically, they adapt the system so that it can maximize either the *conditional* probability of the tree given sentence ($P(C = Y|F = X)$, their *discriminative* RNNG model), or the *joint* probability of the tree and the sentence, ($P(C = Y, F = X)$, their *generative* RNNG model). To do this, they replace the shift transition—which moves a token from the buffer to the stack—with a generate action—which creates a new token and puts it on the stack. Then, to find the tree Y that maximizes the joint probability for a sentence X, they take a trained discriminative model, sample trees for X according to the distribution it generates, and evaluate them according to the generative model. Kuncoro et al. (2016) use the generative RNNG for dependency parsing, as well as ablated variants of it that operate without one of the three LSTMs (stack, buffer, actions) and one that only uses the the stack. The baseline RNNG and the stack-only RNNG both currently hold state-of-the performance

on the popular Stanford Dependencies Penn Treebank benchmark among systems without outside unsupervised data.

4.3 Arc-factored parsing

4.3.1 The algorithm

In a transition-based system, the swap action allows for arbitrary non-projective trees, and the arc-swift system brings both the number of transitions needed per sentence and the number of actions available per transition closer to the length of the sentence. An alternative parsing paradigm to transition-based parsing takes these properties to their logical conclusion, allowing for fully non-projective trees where each word considers all other words when making edge decisions. The core of this thesis will utilize this alternative paradigm, known as *graph-based* or *arc-factored* parsing (McDonald et al., 2005). Transition-based systems attain high efficiency by exploiting the similarity between most dependency formalisms and standard constituency formalisms; arc-factored systems attain high accuracy by exploiting the similarity between most dependency formalisms and graphs.

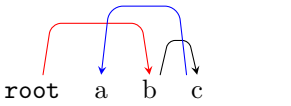
The arc-factored approach scores every possible edge in the dependency tree, and then constrains the resulting weighted graph to have a specific structure. This kind of parsing algorithm is sometimes called graph-based because it treats dependency representations as graphs, and sometimes called arc-factored because the probability of a dependency tree is represented as the probability of its individual arcs (this thesis prefers the latter terminology). Here, if a dependency tree is represented with two matrices—an adjacency matrix A where token t's head word $\tilde{t}$ is indicated with a 1 at $a_{t\tilde{t}}$, and a class matrix C with one column for each label such that word t's label k is indicated with a 1 at c_{tk}—and if a sentence is represented with a feature vector $\mathbf{f}$, then the probability of the whole tree decomposes into the probability of each labeled edge in the tree, shown in Eq. (4.10). This amounts to assuming conditional independence between each labeled edge probability.

$$P(C, A|\mathbf{f}) = \prod_{t=1}^{T} [P(\mathbf{c}_t, \mathbf{a}_t|\mathbf{f})] \tag{4.10}$$

The arc-factored approach—like the transition-based approach—constructs trees using partial rules, but does so more explicitly. Each possible rule "$w_t \rightarrow w_{\tilde{t}\neq t}$" is assigned a probability, and a maximum spanning tree algorithm selects the rules that result in the most probable valid tree. Generally, the whole tree is generated simultaneously. This means features based on partially-built parses are impossible, but otherwise the approach can use the same information available to transition-based algorithms.

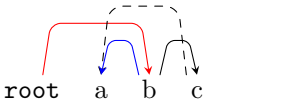
While transtion-based parsers were originally used alongside handcrafted grammars and then adapted to take advantage of feature-based machine learning, graph-based parsers have used machine

learning since their conception. In McDonald et al.'s (2005) parser, first a linear feature-based scorer assigns a score to each edge in the graph, and then one of two *maximum spanning tree* (MST) algorithms is applied to find the connected tree structure with the highest weight. The Chu-Liu/Edmonds (Chu and Liu, 1965; Edmonds, 1967) algorithm can be used to find the highest-scoring non-projective tree, or—if the dependency formalism is known to prohibit crossing dependencies—Eisner's algorithm (Eisner, 1996) can be employed to find the highest-scoring projective tree. Figure 4.5 shows the matrix S of edge scores for a hypothetical string, where each entry $s_{t\tilde{t}}$ represents the score of token t depending on token $\tilde{t}$.



	root	a	b	c
a	.1	0	.5	1
b	1	.1	0	.1
c	.2	.1	1	0

(a) Chu-Liu/Edmonds.



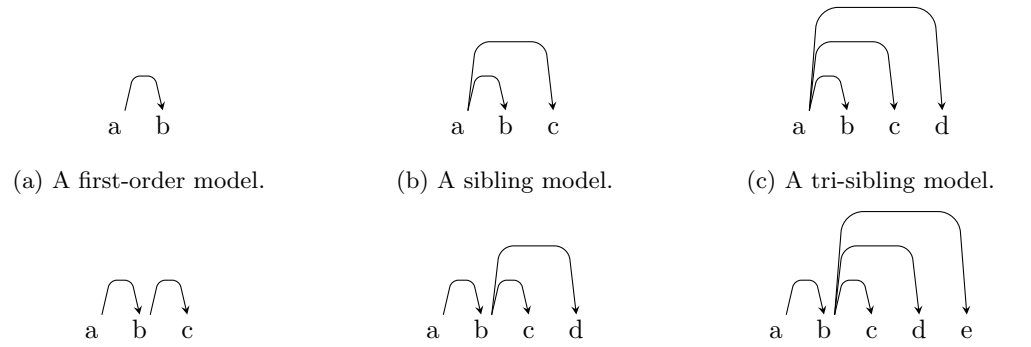
	root	a	b	c
a	.1	0	.5	1
b	1	.1	0	.1
c	.2	.1	1	0

(b) Eisner's.

Figure 4.5: Dependency charts using different maximum spanning tree algorithms.

In Figure 4.5a, the graph scores are parsed into the highest-scoring tree. In Figure 4.5b, the graph scores are parsed into the highest-scoring *non-projective* tree, which may not be the globally maximal one. Both algorithms have a $O(n^3)$ worst case bound, although $O(n^2)$ modifications to the Chu-Liu/Edmonds algorithm exist (Tarjan, 1977; Gabow et al., 1986). As McDonald et al. (2005) point out, ensuring projectivity in a graph-based parser introduces more overhead than permitting nonprojectivity; this is in contrast to transition-based parsers, for which projectivity is easy to ensure and hard to relax. However, the most highly optimized graph-based parsers will clearly never have a time complexity below $O(n^2)$, because each entry in S must be examined at least once. Since transition-based parsers can guarantee a runtime linear in the length of the sentence for projective trees (which are by far more common in natural language (Nivre, 2009)), and rarely approach the quadratic upper bound for non-projective ones, they have historically been more popular than graph-based parsers. This has begun to change for large-scale text parsing in recent years with the rise of neural machine learning algorithms, because neural arc-factored systems—unlike transition-based systems—can easily parse large numbers of sentences in parallel on machines with GPU acceleration or efficient linear algebra subprogram libraries.

The transition-based approach makes weaker independence assumptions than the arc-factored model, allowing it to prune away parses that can't be maximal for a more efficient runtime. While arc-factored parsers, on the other hand, don't run the risk of pruning a correct parse prematurely, the conditional independence assumptions it makes are *too* strong, because edge probabilities are



(a) A first-order model. (b) A sibling model. (c) A tri-sibling model.

(d) A grandparent model. (e) A grand-sibling model. (f) A grand-tri-sibling model.

Figure 4.6: Various higher-order arc-factored models.

not guaranteed to be independent. For example, there may be two likely candidates w_i, w_j for the root of the sentence (the single word depending on the special `root` token, w_0) given the features of the sentence $\mathbf{f}$—$P(w_0 \rightarrow w_i|\mathbf{f})$ and $P(w_0 \rightarrow w_j|\mathbf{f})$—are both relatively high; however, the joint probability $P((w_0 \rightarrow w_i), (w_0 \rightarrow w_j)|F)$ will be zero for most dependency schemes because every sentence must have a unique root. To address this, some work has aimed to construct *higher-order* arc-factored parsers (McDonald and Pereira, 2006). Rather than predicting a single edge, these systems aim to predict *multiple related* edges simultaneously. In a sibling model, the system will aim to predict pairs of edges with a shared parent, and in a grandparent model (Carreras, 2007), the system aims to predict a word's parent and its grandparent. These can be combined and extended, as shown in Figure 4.6. These make it easier to incorporate more context into arc decisions, and it helps to avoid bizarre parses that the first-order maximum spanning tree algorithm isn't able to clean up during postprocessing. However, rather than generating a well-behaved, two-dimensional score matrix, this approach generates a higher-order tensor of partial scores that must be searched through exhaustively to find the highest-scoring tree. That is, in a third-order model, there will be n^3 scores that must all be examined at least once in order to guarantee that the resulting tree is maximal (Koo and Collins, 2010). What's more, exact algorithms that achieve this bound only exist for projective parses. While it is possible to decode a second-order score tensor into a *graph* in cubic time, finding the maximum spanning non-projective *tree* from a higher-order score tensor is NP-hard (McDonald and Pereira, 2006). One of the biggest advantages of arc-factored parsers over transition-based ones is that non-projective trees are easy to predict, so any extension that cripples an arc-factored system's ability to generate nonprojectivity is arguably self-defeating. Fortunately, this intractability can be alleviated by approximate decoding algorithms. McDonald and Pereira (2006) propose an approximate non-projective algorithm that starts with a tree generated using a second-order projective decoder and searches through the second-order scores for non-projective changes that result in a higher-scoring tree; this can be done without increasing the $O(n^3)$ runtime.

Koo et al. (2010) use dual decomposition to combine predictions from a first-order tree decoder and a higher-order graph decoder, adjusting the underlying scores of each until they agree (or mostly agree) on a spanning tree. With a second-order scorer, this algorithm takes a manageable $O(Kn^3)$ time, where K is the maximum number of adjustments. To conclude, higher-order scorers allow the system to compensate somewhat for the faulty independence assumptions in the basic first-order model, at the cost of additional postprocessing overhead. Recent work on arc-factored dependency parsing has focused on exploring what performance can be achieved with a neural architecture coupled with a simple, first-order decoder, leaving higher-order neural approaches waiting to be rediscovered.

4.3.2 Neural arc-factored models

A fairly sizeable number of neural arc-factored models were proposed at roughly the same time, with varying degrees of knowledge about each other. The work at the core of this thesis is among them, so this final section aims to characterize their differences.

Kiperwasser and Goldberg (2016) and Zhang et al. (2017) build the first neural arc-factored parsers with little substantial difference between them. First they feed word and part-of-speech embeddings into a bidirectional LSTM, and then they use traditional feedforward attention over the recurrent states to generate a score for each (ordered) pair of words. They train the attention mechanism to maximize the score between a word and its head and minimize the score between that word and all other words. The resulting matrix of scores can be decoded in a first-order maximum spanning tree parser to produce an unlabeled parse. The labeler works analogously; the recurrent states for a word and its gold (at training time) or predicted (at inference time) head are concatenated and used in a feedforward classifier to predict the label for the edge from the head to the dependent. Conditioning on the head word allows the system to ensure consistency between edges and labels. Zhang et al. (2017) achieve slightly higher performance, likely owing to having a better hyperparameter configuration.

Hashimoto et al. (2017) build a multitask BiLSTM system that includes dependency parsing as one of the tasks. They use the lower layers of their system to do lower-level tasks such as part-of-speech tagging and noun phrase chunking, and then use the higher layers for higher-level tasks, such as parsing and semantic relatedness. Instead of using part-of-speech tag embeddings directly, they use the weighted average of all tags where the weights are the probability the system assigns each tag (similar to (Alberti et al., 2015)). They feed the weighted average tag embedding (as well as a weighted average phrase chunk embedding) into the BiLSTM layer that predicts the dependency parse. Their dependency parsing model is similar to Kiperwasser and Goldberg's, except they use a (shallow) bilinear attention classifier instead of a feedforward one to make edge predictions (otherwise their label classifier is identical). They find that their multitask approach does much better than their single-task approach, beating Andor et al. (2016), who claimed the highest-performing system at the time of publication. However, because of the multitask setup, their system has access to more

data than the competing systems (though not annotated with dependency parses), which mildly confounds their results.

Of the contemporaneous dependency parsing algorithms described in this chapter, Cheng et al. (2016) stands out as being the most innovative. As mentioned previously in this section, the simplest arc-factored parsers—first-order ones—incorrectly assume conditional independence between all arcs. Instead of introducing higher-order scoring with all the additional overhead it comes with, Cheng et al. work around this by keeping track of previous predictions. In their system, each word is assigned a "head" representation with a straightforward BiGRU over part-of-speech and token embeddings. "Dependent" representations are generated twice—once with a unidirectional GRU over the original sequence, and once with a GRU over the reversed sequence. In each direction, the dependent representation assigns every possible head a score using directly-optimized feedforward attention. The scores for each token in each direction are turned into a probability distribution, which is used to compute the weighted average of the head representations. This weighted average is then concatenated with the next input token at the following step of the GRU; thus the dependent representations at each step are fed the sequence of tokens *and* the sequence of previous soft predictions.

$$\tilde{\mathbf{h}}_{\tilde{t}} = \text{BiGRU}_{\tilde{t}}(X) \tag{4.11}$$

$$\mathbf{h}_t = \text{GRU}_t(X_{1:t} \oplus \bar{H}_{0:t-1}) \tag{4.12}$$

$$s_{t\tilde{t}} = \text{FFNN}(\mathbf{h}_t, \tilde{\mathbf{h}}_{\tilde{t}}) \tag{4.13}$$

$$\mathbf{a}_t = \text{softmax}(\mathbf{s}_t) \tag{4.14}$$

$$\bar{\mathbf{h}}_t = \tilde{H}_{1:t-1}^{\top} \tilde{\mathbf{a}}_t \tag{4.15}$$

During inference, the forward and backward scores are added together before being used in a greedy argmax decoder (which doesn't guarantee a well-formed tree) or in Eisner's projective MST decoder. This allows the system to condition predictions on previous predictions, although the authors must sacrifice the representational capacity of the dependent representation, which cannot generate representations that take advantage of bidirectional information. These representations are re-used in a feedforward classifier to compute the labels; however, whereas other approaches use the gold or predicted head's recurrent vector, this one uses the weighted average head vectors. Unfortunately, this introduces the possibility of generating labels that are inconsistent with the selected head, which the authors don't address. This approach achieves the highest performance of the arc-factored systems they compare against, although it is still beaten by the transition-based system of Andor et al. (2016).

4.4 Conclusion

Both transition-based and arc-factored parsers have in common that they don't rely on strict conjunctions of features or complex underspecified categories to generate parse trees. This allows them to guarantee at least one valid parse, because they don't have to worry about lacking a necessary rule. Because they use learned classifiers, they produce a probability distribution over all possible trees, allowing them to thrive in the face of spurious ambiguity and return only the parse they're most confident in. This makes them accurate, practical, and reasonably efficient. Furthermore, they make only minimal sacrifices to linguistic theory. These parsing approaches still learn the tree structure rules ubiquitous in syntactic theory—they just break the rules down into smaller and more manageable pieces, which helps to find and take advantage of patterns among similar but non-indentical syntactic structures (which linguistic theories arguably struggle to do elegantly). Additionally, while explicit unification is unrealistic in these systems, features indicating that relevant words or phrases have incompatible values can be used to achieve similar effects. These features can be explicitly hand-engineered (e.g. `person-disagr`), or (less efficiently) generated through conjunctions of simplex features (e.g. `N-sg&V-pl`). Finally, the strategy of entertaining all possible tree structures and selecting the one with the least entropy bears a clear relationship to Harmonic Grammar (Legendre et al., 1990), which is compatible with constraint-based approaches to syntax (such as HPSG and LFG; see Chapter 2) and which developed out of connectionism and artificial neural networks. In a Harmonic Grammar, all output candidates for some input are considered and assigned weighted penalties based on which linguistic constraints (such as "must have a subject") they violate, and how badly they violate them. The output candidate with the smallest accumulated penalty weight is selected.

The work in this thesis adds to the literature on arc-factored dependency parsing. Neural transition-based systems have a number of advantages that stem from their close relationship to constituency tree parsers, such as an efficient $O(n)$ runtime for projective trees and the ability to take advantage of partially-built structures. However, they have some notable drawbacks: they require extra complexity to avoid propagating errors through the transition sequence (through dynamic oracles or the arc-swift transition system); they require extra complexity to capture non-projective dependencies (through additional data structures or transition actions); as will be seen in Section 7, they require even more complexity to generate arbitrary, non-tree graphs; and they cannot be batched efficiently in neural systems. By contrast, while arc-factored parsers cannot currently take advantage of partially-built trees without complex higher-order structures (or at least, not without other sacrifices in the case of Cheng et al. 2016) and have slightly worse theoretical runtime complexity, they can be used to parse a sentence in a projective tree, a non-projective tree, or an arbitrary graph with relative ease.

The arc-factored systems described in this chapter almost exclusively make use of feedforward classifiers instead of the biaffine classifiers motivated theoretically in Chapter 3. Additionally, they

lag behind the state-of-the-art transition-based system by a very wide margin. Thus Chapter 5 will compare the feedforward classifier to a comparable biaffine one both theoretically and empirically, finding that the biaffine approach is more mathematically sensible and outperforms the traditional feedforward one. Additionally, the system hyperparameters will be explored extensively, in order to bring performance closer to what has been reported for transition-based parsers. Chapter 6 will then propose a number of extensions to the baseline system and compare the ability of the arc-factored model against a transition-based baseline. Finally, Chapter 7 will show how the system can be straightforwardly tweaked to produce arbitrary directed labeled graphs instead of trees, demonstrating the power and flexibility of the arc-factored approach.

Chapter 5

Biaffine Dependency Parsing

5.1 Introduction

This chapter describes the crux of this thesis, namely the neural approach to dependency parsing laid out by Dozat and Manning (2017). It was developed concurrently with several other, related approaches, which will be first described in Section 5.2. The architecture of the proposed model will be motivated theoretically in Section 5.3 and empirically in Section 5.4. Some extensions by other researchers are laid out in Section 5.5; extensions by this researcher are detailed in subsequent chapters of this thesis.

The current state-of-the-art transition-based neural dependency parser (Kuncoro et al., 2016) substantially outperforms many much simpler neural arc-factored (i.e. graph-based) parsers. Dozat and Manning (2017) use a variant of the neural arc-factored approach first proposed by Zhang et al. (2017) and Kiperwasser and Goldberg (2016) but achieves competitive performance to the transition-based state-of-the-art: the network is larger but uses more regularization; it replace the traditional attention mechanism and feedforward network (FFNN) label classifier with biaffine ones; and rather than using the top recurrent states of the LSTM in the biaffine transformations, it first puts them through FFNN operations that reduce their dimensionality. This has a number of advantages over previous approaches: it's very simple, consisting of only linear algebraic operations and standard neural nonlinearities; multiple sentences can be batched efficiently; and in spite of this simplicity, it achieves state-of-the-art or near state-of-the-art accuracy on all standard benchmarks. Because of its relative simplicity, it can be fairly easily extended with additional complexity. Furthermore, because of its high performance that resulted from careful tuning, any extentions that yield higher performance are likely to represent real architectural or algorithmic improvements, and are probably not simply compensating for suboptimal hyperparameter choices. Dozat and Manning (2017) motivate two new kinds of classifiers: one for when the number of classes isn't fixed, and one for when the prediction depends on two interacting input sources. These new simple and straightforward

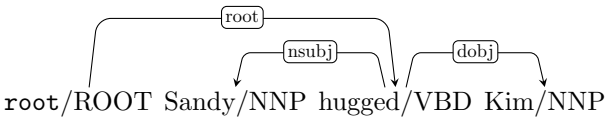


Figure 5.1: A simple dependency tree parse for *Sandy hugged Kim*. Includes part-of-speech tags and a special `root` token. Directed edges (or arcs) with labels (or relations) connect the verb to the root and the arguments to the verb head.

classifier types can be used in networks designed for other tasks beside dependency parsing.

5.2 Background and Related Work

Arc-factored parsers use machine learning to assign a weight or probability to each possible edge and then construct a maximum spaning tree (MST) from these weighted edges. Zhang et al. (2017) and Kiperwasser and Goldberg (2016) concurrently present a neural arc-factored parser (in addition to a transition-based one) that uses the same kind of attention mechanism as Bahdanau et al. (2014) for machine translation. In their models, the (bidirectional) LSTM's output state for each word is concatenated with each possible head's output state, and the result is used as input to an MLP that scores each resulting arc. The predicted tree structure at training time is the one where each word depends on its highest-scoring head. Labels are generated analogously, with each word's LSTM output vector and its gold or predicted head word's output vector being used in a multi-class FFNN.

Similarly, Hashimoto et al. (2017) include a graph-based dependency parser in their multitask neural model. In addition to training the model with multiple distinct objectives, they replace the traditional FFNN-based attention mechanism that Kiperwasser and Goldberg (2016) use with a bilinear one (but still using an FFNN label classifier). This makes it analogous to Luong et al.'s 2015 proposed attention mechanism for neural machine translation. Cheng et al. (2016) likewise propose a graph-based neural dependency parser, but in a way that attempts to circumvent the limitation of other neural graph-based parsers being unable to condition the scores of each possible arc on previous parsing decisions. In addition to having one bidirectional recurrent network that computes a recurrent hidden vector for each word, they have additional, unidirectional recurrent networks (left-to-right and right-to-left) that keep track of the probabilities of each previous arc, and use these together to predict the scores for the next arc.

5.3 Proposed Dependency Parser

5.3.1 Basic architecture

This work is very similar to the arc-factored architectures of Kiperwasser and Goldberg (2016), Hashimoto et al. (2017), and Cheng et al. (2016), shown in Figure 5.2. Like their systems, the proposed one uses BiLSTMs over concatenated word and POS tag embeddings. The first difference involves using biaffine attention to predict the dependency edges instead of bilinear or traditional FFNN-based attention. The second difference involves using biaffine classifiers to predict the dependency labels instead of the usual neural classifer. Additionally, both modules are made faster and less prone to overfitting by reducing the size of the recurrent states with smaller feedforward layers before using them in the classifiers. Finally, the system uses a more carefully trained hyperparameter configuration, with larger hidden states but more regularization.

The notation in this section will follow the same convention described at the beginning of Chapter 3. Here, the tokens in sentence i will be represented as a sequence of T_i vector-space embeddings $(\mathbf{x}_{i,1}, \ldots, \mathbf{x}_{i,T})$, stacked into a matrix X_i. For notational compactness, the i subscript will be left off. These embeddings can be simple word embeddings in a lookup table, or they can be contextualized with more complex neural systems. A parse tree will be a pair of matrices, Y and $\tilde{Y}$: each row t of Y is a one-hot vector encoding the dependency label of word t, and each row t of $\tilde{Y}$ is a one-hot vector encoding the position of the head for word t. In general, variables with tildes will relate to heads (or potential heads), and unannotated variables will relate to dependents.

In an arc-factored dependency parser, the objective is to estimate the probability of all possible labeled parses for a given sentence X, and select the most likely parse subject to any well-formedness constraints (which may block cycles or multiple roots). Formally, the goal is to compute Eq. (5.1):

$$\arg\max_{Y,\tilde{Y}} \Big[P(C = Y, A = \tilde{Y} | F = X) \Big] \tag{5.1}$$

That is, to compute the most probable dependency relations (or *Classes*, hence the C variable) and the most probable dependency heads (or *Arcs*, hence the A variable; A can also be thought of as an *Attention* or directed *Adjacency* matrix) given the token features F. Arc-factored parsers, as the name suggests, simplify this objective by assuming conditional independence between arcs in order to factorize the problem (much like Naïve Bayes or Maximum Entropy classifiers). The features for the token seeking its head will be represented with the $\mathbf{f}$ variable, and the features for the other tokens in the sentence that it can attend to will be in $\tilde{F}$.

$$P(C = Y, A = \tilde{Y} | F = X) = \prod_{t=1}^{T} \prod_{\tilde{t}=1}^{T} \prod_{k=1}^{m} \Big[P(c_{tk} = 1, a_{t\tilde{t}} = 1 | \mathbf{f} = \mathbf{x}_t, \tilde{F} = X)^{y_{tk}\tilde{y}_{t\tilde{t}}} \Big] \tag{5.2}$$

Exponentiating by y_{tk} and $\tilde{y}_{t\tilde{t}}$ ensures that only the probabilities of the labeled edges that the tree

actually assumes are accumulated into the product. This is sensible because each row of Y and $\tilde{Y}$ is categorically distributed. Thus in Eq. (5.2), the probability of the whole tree $(Y, \tilde{Y})$ is represented as a product of the conditional probabilities of all labeled edges in the tree. The assumption of conditional independence is obviously incorrect, since a head with a nominal dependency label is particularly unlikely to assign labeled edges that only verbs can assign, like subjct or object. Furthermore, cycles are normally prohibited, such that the joint probability of token i depending on token j and token j depending on token i is zero, even when the marginal probabilities of the two dependencies are nonzero. However, it's normally "good enough" to achieve high performance. $c_{tk} = 1$ and $a_{t\tilde{t}} = 1$ will be shortened to c_{tk} and $a_{t\tilde{t}}$ henceforth, and similarly for $\mathbf{f}$ and $\tilde{F}$.

The joint conditional probabilities in Eq. (5.2) will be easier to work with if they're *factorized* into two parts, each being the conditional probability of a single variable. For further notational simplicity, the t index of the C and A variables will be dropped.

$$P(c_k, a_{\tilde{t}}|\mathbf{f}, \tilde{F}) = \frac{P(c_k, a_{\tilde{t}}, \mathbf{f}, \tilde{F})}{P(\mathbf{f}, \tilde{F})} \tag{5.3}$$

$$= \frac{P(c_k, a_{\tilde{t}}, \mathbf{f}, \tilde{F})}{P(\mathbf{f}, \tilde{F})} \frac{P(a_{\tilde{t}}, \mathbf{f}, \tilde{F})}{P(a_{\tilde{t}}, \mathbf{f}, \tilde{F})} \tag{5.4}$$

$$= \frac{P(c_k, a_{\tilde{t}}, \mathbf{f}, \tilde{F})}{P(a_{\tilde{t}}, \mathbf{f}, \tilde{F})} \frac{P(a_{\tilde{t}}, \mathbf{f}, \tilde{F})}{P(\mathbf{f}, \tilde{F})} \tag{5.5}$$

$$= P(c_k|\mathbf{f}, \tilde{F}, a_{\tilde{t}}) P(a_{\tilde{t}}|\mathbf{f}, \tilde{F}) \tag{5.6}$$

$$= P(c_k|\mathbf{f}, \tilde{f}_{\tilde{t}}, a_{\tilde{t}}) P(a_{\tilde{t}}|\mathbf{f}, \tilde{F}) \tag{5.7}$$

Eq. (5.3) expresses the conditional probability as a ratio of joint probabilities. Eq. (5.4) multiplies by $\frac{z}{z}$, and Eq. (5.5) switches the denominators. Eq. (5.6) converts the ratios back to conditional probabilities. This process maintains exact equivalence without assuming conditional independence between the probability of word t's arc and the probability of its label. Eq. (5.7) then simplifies the conditional probability of the label by assuming conditional independence between the label and the features of tokens other than $\tilde{f}_{\tilde{t}}$, given that $a_{\tilde{t}} = 1$. This two-module setup allows for simpler decoding; if the most probable parse according to the model is an invalid tree, the edges can be rearranged independent of the labels. Then, after a valid parse has been settled on, labels can be assigned greedily without changing the optimality of the structure.

The next step is to decide on classifiers for each of these probabilities. To briefly recapitulate, Chapter 3.1 proves that the probability of a class given a set of binary-valued features $P(c_k = 1|\mathbf{f} = \mathbf{x}; \theta)$ is exactly equivalent to a Maximum Entropy, or *affine softmax classifier*, assuming all features are conditionally independent given $c_k = 1$. That is, the lefthand side of Eq. (5.9) can be simply rewritten into the righthand side, maintaining exact equivalence, for some assignment of parameters

W and $\mathbf{b}$.

$$\text{Aff}(\mathbf{x}) = W\mathbf{x} + \mathbf{b} \tag{5.8}$$

$$P(c_k|\mathbf{f} = \mathbf{x}; \theta) = \text{softmax}_k(\text{Aff}(\mathbf{x})) \tag{5.9}$$

The weights and biases in θ must still be optimized in order to minimize the cross-entropy of a given training corpus, but the expression in Eq. (5.9) is provably correct under conditional independence. Chapter 3.2 then goes on to show that the assumption of conditional independence can be relaxed at the cost of increasing the parameters in the classifier. Using a biaffine softmax classifier instead of an affine one adds interaction terms, which is appropriate if the features are *pairwise* conditionally dependent (Eq. 5.11). It can also be used when there are two sets of features, $\mathbf{f}$ and $\tilde{\mathbf{f}}$, that are internally conditionally independent but pairwise conditionally dependent with each other (Eq. 5.12).

$$\text{Biaff}(\mathbf{x}, \tilde{\mathbf{x}}) = \mathbf{x}^\top \mathbf{U}\tilde{\mathbf{x}} + W(\mathbf{x} \oplus \tilde{\mathbf{x}}) + \mathbf{b} \tag{5.10}$$

$$P(c_k|\mathbf{f} = \mathbf{x}; \theta) = \text{softmax}_k(\text{Biaff}(\mathbf{x}, \mathbf{x})) \tag{5.11}$$

$$P(c_k|\mathbf{f} = \mathbf{x}, \tilde{\mathbf{f}} = \tilde{\mathbf{x}}; \theta) = \text{softmax}_k(\text{Biaff}(\mathbf{x}, \tilde{\mathbf{x}})) \tag{5.12}$$

When the classes vary from example to example—such as when the available classes are positions in a sentence—the input variables *must* interact with features $\tilde{F}$ of the class (Eq. 5.14). Eq. (5.14) is then a *variable-class* biaffine softmax classifier, as opposed to the *fixed-class* biaffine softmax classifiers in Eqs. (5.11, 5.12).

$$\text{VCBiaff}(\mathbf{x}, \tilde{X}) = \tilde{X}U\mathbf{x} + \tilde{X}\mathbf{w} \tag{5.13}$$

$$P(c_k|\mathbf{f} = \mathbf{x}, \tilde{F} = \tilde{X}; \theta) = \text{softmax}_k(\text{VCBiaff}(\mathbf{x}, \tilde{X})) \tag{5.14}$$

Returning now to dependency parsing, the first goal is to identify the location of word t's head in sentence i; since the number of positions varies from sentence to sentence, a variable-class classifier like the one in Eq. (5.14) is a logical starting point. The second goal is to identify the label that the head assigns the dependent; however, this label may depend on both features of the dependent and features of its head. It's reasonable to hypothesize that the two sets of features may interact, such that $P(c_k|\mathbf{f}, \tilde{\mathbf{f}}) \neq P(c_k|\mathbf{f})P(c_k|\tilde{\mathbf{f}})$. This suggests that a fixed-class biaffine classifier as in Eq. (5.12) is appropriate, with one source of features coming from the dependent and another from the head. The *a priori* decision to use biaffine classifiers is also emprically motivated, as Chen and Manning (2014) found that feature conjunctions were critical for achieving high performance in feature-based transition parsers. Observing this, they attempted to implicitly build three-way feature conjunctions into their system with what amounts to a decomposed triaffine transformation of the input embeddings. Biaffine transformations explicitly include two-way feature conjunctions, lending further credence to the decision to use them as the basis for the dependency classifiers.

Given that the parser will be using these biaffine classifiers, how should the input features be derived? The current neural architecture of choice for dealing with sequences—especially sentences—is the bidirectional long short-term memory network (BiLSTM). The input to the BiLSTM will be concatenated word and POS tag embeddings, following other work on dependency parsing.

$$\mathbf{e}_t = \mathbf{e}_t^{(\text{word})} \oplus \mathbf{e}_t^{(\text{pos})} \tag{5.15}$$

$$\mathbf{h}_t = \text{BiLSTM}_t(E) \tag{5.16}$$

The simplest approach would be to use the LSTM vector $\mathbf{h}_t$ as the input feature vector and the LSTM states of the whole sentence as the context vectors into Eqs. (5.12, 5.14).

$$P(a_{\tilde{t}}|\mathbf{f} = \mathbf{h}_t, \tilde{F} = H) = \text{softmax}_{\tilde{t}}(\text{VCBiaff}(\mathbf{h}_t, H)) \tag{5.17}$$

$$\hat{\tilde{t}}_t = \arg\max_{\tilde{t}} \Big[P(a_{\tilde{t}}|\mathbf{f}, \tilde{F})\Big] \tag{5.18}$$

$$P(c_k|\mathbf{f} = \mathbf{h}_t, \tilde{\mathbf{f}} = \mathbf{h}_{\hat{\tilde{t}}_t}) = \text{softmax}_{\tilde{t}}(\text{Biaff}(\mathbf{h}_t, \mathbf{h}_{\hat{\tilde{t}}_t})) \tag{5.19}$$

This approach can be tweaked slightly in order to make the system both faster and more accurate. It can be intuited that the recurrent vectors H must contain a lot of information: they need to contain features used for identifying word t's head, all and only t's dependents, the best label for t given any of the likely head words, the best label of all of t's possible dependents, and any recurrent features that may be needed in other parts of the sentence. This is clearly more information than is needed for any individual classification objective. Ideally, the recurrent vectors H should be large enough to contain the necessary information, but the information flow into the classifiers should be restricted in order to reduce overfitting. This can be done by applying separate FFNNs (each with distinct parameters) to the recurrent vectors before using them in the classifiers. If the FFNNs have smaller hidden states than the RNN, then the network will be forced to strip away information not necessary for the current classification task. This will also likely increase the system's speed, since the time complexity of the classifiers is quadratic in the size of the hidden state. Interestingly, other researchers have found that using FFNNs to split the LSTM state into specialized representations can be beneficial for other tasks, such as Reed and de Freitas (2016); Miller et al. (2016); Daniluk et al. (2017). Inserting the FFNNs adds depth to the classifiers, so these new functions will be referred to as *deep* biaffine softmax classifiers.

$$\text{DeepVCBiaff}(\mathbf{x}, X) = \text{VCBiaff}\big(\text{FFNN}(\mathbf{x}), \text{FFNN}(X)\big) \tag{5.20}$$

$$\text{DeepBiaff}(\mathbf{x}, \tilde{x}) = \text{Biaff}\big(\text{FFNN}(\mathbf{x}), \text{FFNN}(\tilde{\mathbf{x}})\big) \tag{5.21}$$

The whole algorithm for the proposed dependency parser, including the summed cross-entropy loss objective for a single example, is summarized below. Figure 5.2 provides a visual representation of

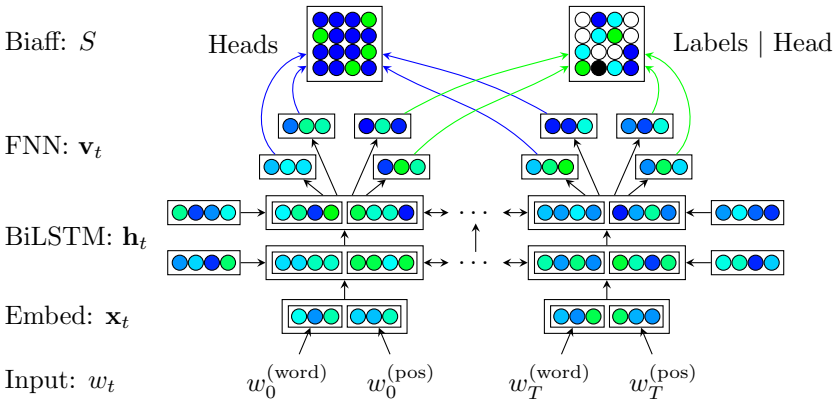


Figure 5.2: The basic parser architecture.
BiLSTM with deep biaffine attention to first score each possible head for each dependent, and then to score each possible label for each possible head.

the system.

$$\mathbf{e}_t = \mathbf{e}_t^{(\text{word})} \oplus \mathbf{e}_t^{(\text{pos})} \tag{5.22}$$

$$\mathbf{h}_t = \text{BiLSTM}_t(E) \tag{5.23}$$

$$\hat{\tilde{\mathbf{y}}}_t = \text{softmax}(\text{DeepVCBiaff}(\mathbf{h}_t, H)) \tag{5.24}$$

$$\hat{\tilde{t}}_t = \arg\max\left[\hat{\tilde{\mathbf{y}}}_t\right] \tag{5.25}$$

$$\hat{\mathbf{y}}_t = \text{softmax}(\text{DeepBiaff}(\mathbf{h}_t, \mathbf{h}_{\tilde{t}_t})) \tag{5.26}$$

$$\varepsilon = -\sum_{t=1}^{T}\left[\mathbf{y}_t \ln(\hat{\mathbf{y}}_t) + \tilde{\mathbf{y}}_t \ln(\hat{\tilde{\mathbf{y}}}_t)\right] \tag{5.27}$$

5.3.2 Comparison with traditional attention

One of the key insights in this work is the biaffine approach to classification, in particular variable-class classification. Countless systems already utilize variable-class classifiers as latent variables in neural networks, though generally not with the architecture laid out above. In such cases, they are known as *attention* or *attention mechanisms*. Normally a weighted sum of the word representations H is used, where each weight is a probability produced by some attention function, which is functionally a variable-class classifier.

$$\mathbf{a}_t = \text{Attn}(\mathbf{h}_t, \tilde{H}) \tag{5.28}$$

$$\bar{\tilde{\mathbf{h}}}_t = \tilde{H}^\top \mathbf{a}_t \tag{5.29}$$

This is known to improve sequence generation tasks such as Neural Machine Translation (Bahdanau et al., 2014). Most often, the Attn function is a feedforward network that takes as input a hidden "attender" vector $\mathbf{h}_t$ and a hidden "attendee" vector $\tilde{\mathbf{h}}_{\tilde{t}}$, and produces a score $s_{t\tilde{t}}$ for the pair. The score vector $\mathbf{s}_t$ is then normalized with softmax and used as the weights for the weighted average. The position of this thesis is that the variable-class biaffine classifier or attention is more principled and intuitive than the variable-class feedforward one. The previous chapter demonstrated the relationship between affine and biaffine classifiers, as well as their joint relationship to information theory. Here it will be shown that the variable-class feedforward classifier lacks these appealing properties. The variable-class feedforward scorer and analogous biaffine scorer are shown in Eqs. (5.30, 5.31).

$$s_{t\tilde{t}} = \mathbf{u}^\top f(W\mathbf{h}_t + \tilde{W}\tilde{\mathbf{h}}_{\tilde{t}} + \mathbf{b}) \tag{5.30}$$

$$s_{t\tilde{t}} = \tilde{\mathbf{h}}_{\tilde{t}}^\top U\mathbf{h}_t + \tilde{\mathbf{h}}_{\tilde{t}}^\top \mathbf{w} \tag{5.31}$$

$$\mathbf{a}_t = \text{softmax}(\mathbf{s}_t) \tag{5.32}$$

The feedforward approach is used to solve the problem raised in Chapter 3.2.2, that some form of nonlinear interaction between the attender features $\mathbf{f}$ and the attendee features $\tilde{F}$ is needed for the classifier to produce meaningful predictions. Instead of using multiplicative interactions, it uses pseudo-additive interactions; the features are weighted separately, added, and then "squashed" with tanh or ReLU. These can be thought of as "pair" features, representing the pair of words together. The pair features are then used in a linear discriminator, which weights each pair (using the same weight vector) to decide on the best one.

If, hypothetically, the attendee feature vectors $\tilde{H}$ were to be the same over all examples, then the biaffine scorer in Eq. (5.31) would reduce to an affine function with $W = \tilde{H}U$ and $\mathbf{b} = \tilde{H}\mathbf{w}$.

$$\mathbf{w}_k'^\top = \tilde{\mathbf{h}}_k^\top U \tag{5.33}$$

$$b_k' = \tilde{\mathbf{h}}_k^\top \mathbf{w} \tag{5.34}$$

$$s_{tk} = \tilde{\mathbf{h}}_k^\top U\,\mathbf{h}_t + \tilde{\mathbf{h}}_k^\top \mathbf{w} \tag{5.35}$$

$$= \mathbf{w}_k'^\top \mathbf{h}_t + b_k' \tag{5.36}$$

This is ideal; simplifying the task leads to a simpler but still principled model. The feedforward scorer in Eq. (5.30), however, remains a feedforward network, but uses a different bias vector for each class. This is because the class vectors no longer depend on the input, so $\tilde{H}$, their weights $\tilde{W}$,

and the bias term $\mathbf{b}$ can be collapsed into a single variable.

$$\mathbf{b}'_k = \tilde{W}\tilde{\mathbf{h}}_k + \mathbf{b} \tag{5.37}$$

$$s_{tk} = \mathbf{u}^\top f(W\mathbf{h}_t + \tilde{W}\tilde{\mathbf{h}}_k + \mathbf{b}) \tag{5.38}$$

$$= \mathbf{u}^\top f(W\mathbf{h}_t + \mathbf{b}'_k) \tag{5.39}$$

This model, like the variable-class feedforward classifier, uses pseudo-additive interactions between the input and the class to generate pair features that it then weights. However, the structure of the pair features is hardly principled. One could just as well propose a fixed-class feedforward classifier that uses a different weight matrix W_k for each class as well as, or instead of, a different bias vector $\mathbf{b}_k$ (Eq. 5.40), which would have the variable-class analog in Eq. (5.41).

$$s_{tk} = \mathbf{u}^\top f(W_k\mathbf{h}_t + \mathbf{b}_k) \qquad \text{Fixed-class} \tag{5.40}$$

$$s_{t\tilde{t}} = \mathbf{u}^\top f(\tilde{\mathbf{h}}_{\tilde{t}}^\top \mathbf{U}\mathbf{h}_t + W\tilde{\mathbf{h}}_{\tilde{t}}) \qquad \text{Variable-class} \tag{5.41}$$

Alternatively, one could propose conditioning $\mathbf{u}$, not the FFNN, on the class (Eq. 5.43), which would have the variable-class analog in Eq. (5.44).

$$\mathbf{v}_t = \text{FFNN}(\mathbf{h}_t) \tag{5.42}$$

$$s_{tk} = \mathbf{u}_k^\top \mathbf{v}_t \qquad \text{Fixed-class} \tag{5.43}$$

$$s_{t\tilde{t}} = \tilde{\mathbf{h}}_{\tilde{t}}^\top U\mathbf{v}_t \qquad \text{Variable-class} \tag{5.44}$$

Interestingly, both alternative variable-class classifier use bilinear terms, making them effectively deeper and more complex variants of fixed- and variable-class biaffine classifiers. Additionally, all feedforward approaches in Eqs. (5.30, 5.41, 5.44) model only the joint likelihood of the input and class features given $a_{\tilde{t}} = 1$, under the usual conditional independence assumptions. However, they leave out terms relating to the likelihood of the class features given $a_{\tilde{t}} = 1$ *independent* of the input features. That is, some classes may be more likely than others; in many dependency schemes, for instance, $a_{\tilde{t}} = 1$ is more likely if token $\tilde{t}$ is a content word than if it's a function word. The FFNN-based approaches are missing a term for this. It could be added in to Eqs. (5.30, 5.41) by including a second FFNN conditioned only on the class vectors.

$$s_{t\tilde{t}} = \mathbf{u}^{(1)\top} f(W\mathbf{h}_t + \tilde{W}^{(1)}\tilde{\mathbf{h}}_{\tilde{t}} + \mathbf{b}^{(1)}) + \mathbf{u}^{(2)\top} f(\tilde{W}^{(2)}\tilde{\mathbf{h}}_{\tilde{t}} + \mathbf{b}^{(2)}) \tag{5.45}$$

Because of the nonlinearity f, this expression cannot be simplified to use only one FFNN. Fortunately, a bias term can be brought back into Eq. (5.44) without adding another FFNN; however, the end result turns out to be no different from a variable-class biaffine classifier with awkwardly

asymmetrical depth.

$$\mathbf{v}_t = \text{FFNN}(\mathbf{h}_t) \tag{5.46}$$

$$s_{t\tilde{t}} = \tilde{\mathbf{h}}_{\tilde{t}}^\top U \mathbf{v}_t + \tilde{\mathbf{h}}_{\tilde{t}}^\top \mathbf{w} \tag{5.47}$$

Thus the traditional feedforward approach to attention and variable-class classification lacks the clearly principled fixed-class analog that the biaffine approach has. Furthermore, attempts to make the fixed-class feedforward classifier more principled in turn change the variable-class feedforward classifier into something that looks disturbingly similar to a biaffine system.

This comparison shows that there are actually a large number of different architectures that can be used to create a fixed- or variable-class classifier. Of these, it was argued that the affine and biaffine softmax classifiers are the simplest and most principled way to build a classifier for one and two sets of features, respectively.

5.3.3 Hyperparameter configuration

Param	**Value**	**Param**	**Value**
Embedding size	100	Embedding dropout	33%
LSTM size	400	LSTM dropout	33%
Arc MLP size	500	Arc MLP dropout	33%
Label MLP size	100	Label MLP dropout	33%
# LSTM layers	3	# FFNN layers	1
α	2E^{-3}	β_1,β_2	.9
Annealing	$.75^{\frac{t}{5000}}$	$t_{\max}$	50,000

Table 5.1: Basic model hyperparameters.

Aside from architectural differences between Dozat and Manning (2017) and the other arc-factored parsers, this work makes a number of hyperparameter choices that allow the system to outperform theirs, laid out in Table 5.1. The system uses 100-dimensional uncased trainable word vectors, added to 100-dimensional GloVe embeddings (Pennington et al., 2014), concatenated with POS tag vectors; three BiLSTM layers (400 dimensions in each direction); and 500- and 100-dimensional ReLU MLP layers. It also applies dropout at every stage of the model: words and tags are dropped independently; nodes in the LSTM layers (input and recurrent connections) are dropped as well, with the same dropout mask applied at every recurrent timestep (cf. the Bayesian dropout of Gal and Ghahramani (2016)); and nodes in the MLP layers and classifiers are dropped as well, again with the same dropout mask applied at every timestep. The network is optimized with annealed Adam (Kingma and Ba, 2015) for about 50,000 steps, rounded up to the nearest epoch.

Model	UAS	LAS	Sents/sec
Deep	**95.75**	**94.22**	**410.91**
Shallow	95.74	94.00*	298.99
Shallow, 50% drop	95.73	94.05*	300.04
Shallow, 300d	95.63*	93.86*	373.24
MLP	95.53*	93.91*	367.44

Table 5.2: Comparison of classifier architectures. Shows accuracy and speed on PTB-SD 3.5.0. Statistically significant differences are marked with an asterisk.

5.4 Experiments & Results

5.4.1 Datasets

This section shows test results for the proposed model on the English Penn Treebank, converted into Stanford Dependencies using both version 3.3.0 and version 3.5.0 of the Stanford Dependency converter (PTB-SD 3.3.0 and PTB-SD 3.5.0); the Chinese Penn Treebank; and the CoNLL 09 shared task dataset, following standard practices for each dataset. Punctuation was omitted from evaluation only for the PTB-SD and CTB. For the English PTB-SD datasets, the system use POS tags generated from the Stanford POS tagger (Toutanova et al., 2003); for the Chinese PTB dataset it uses gold tags; and for the CoNLL 09 dataset it uses the provided predicted tags. The hyperparameter search was done with the PTB-SD 3.5.0 validation dataset in order to minimize overfitting to the more popular PTB-SD 3.3.0 benchmark, and in the hyperparameter analysis in the following section the reported performance is on the PTB-SD 3.5.0 test set.

5.4.2 Hyperparameter choices

Attention mechanism

Section 5.3 proposes an alternative approach to variable-class classification, dubbed *deep biaffine* attention. This discussion aims to empirically motivate both the added depth of the classifier as well as the biaffine architecture. The FFNN layers allow for large LSTM states while still conserving parameters, which by hypothesis should reduce overfitting and increase inference speed. The variable-class biaffine architecture is arguably more natural than the variable-class FFNN alternative, raising the question of whether it results in observably higher accuracy. Table 5.2 provides evidence supporting these hypotheses. The model with shallow bilinear arc and label classifiers gets the same unlabeled performance as the deep model with the same settings, but presumably because the label classifier is much larger ($(801 \times m \times 801)$ as opposed to $(101 \times m \times 101)$, where m is the number of classes), it runs much slower and overfits. One way to mitigate this overfitting is by increasing the FFNN dropout, but that of course won't influence parsing speed; another way is to decrease the recurrent size to 300, but this only serves to hinder unlabeled accuracy without even

Model	UAS	LAS	Sents/sec
3 layers, 400d	95.75	94.22	410.91
3 layers, 300d	95.82	**94.24**	460.01
3 layers, 200d	95.55*	93.89*	469.45
2 layers, 400d	95.62*	93.98*	**497.99**
4 layers, 400d	**95.83**	94.22	362.09

Table 5.3: Comparison of network sizes.
Test accuracy and speed on PTB-SD 3.5.0. Statistically significant differences are marked with an asterisk.

increasing parsing speed up to the same levels as the deeper model. The traditional FFNN-based approach to attention and classification used in Kiperwasser and Goldberg (2016)[1] was likewise somewhat slower and significantly underperformed the deep biaffine approach in both labeled and unlabeled accuracy. This supports the hypothesis that the deep bilinear architecture is the ideal approach to variable-class classification in terms of both speed and accuracy.

Network size

This discussion examines more closely how network size influences speed and accuracy. Kiperwasser and Goldberg's 2016 model uses 2 layers of 125-dimensional bidirectional LSTMs; Zhang et al.'s 2017 model, uses 2 layers of 300-dimensional BiLSTMs; Hashimoto et al.'s 2017 model uses one layer of 100-dimensional bidirectional LSTMs dedicated to parsing (two lower layers are also trained on other objectives); and Cheng et al.'s 2016 model uses one layer of 368-dimensional GRU cells. This raises the question of how important larger LSTMs are to parsing accuracy. Table 5.3 shows that using three or four layers gets significantly better performance than two layers, and increasing the LSTM sizes from 200 to 300 or 400 dimensions likewise signficantly improves performance.[2] This suggests that the numbers reported in the other systems could be improved simply through larger networks (with more regularization) and further hyperparameter optimization.

Recurrent cell

GRU cells have been promoted as a faster and simpler alternative to LSTM cells, and are used in the approach of Cheng et al. (2016); this discussion aims to weigh in on the debate. The primary difference between GRU cells and LSTM cells, which makes the former in some sense simpler than the latter, is twofold: they combine the input and forget gates into one; and they apply the output gate at the beginning of the cell update rather than at the end, allowing the cell state and output

[1] In the version of TensorFlow used for these experiments, the model's memory requirements during training exceeded the available memory on a single GPU when default settings were used, so the FFNN hidden size was reduced to 200

[2] The model with 400-dimensional recurrent states significantly outperforms the 300-dimensional one on the validation set, but not on the test set

Model	UAS	LAS	Sents/sec
LSTM	**95.75**	**94.22**	410.91
GRU	93.18*	91.08*	435.32
Cif-LSTM	95.67	94.06*	**463.25**

Table 5.4: Comparison of recurrent cell types. Test accuracy and speed on PTB-SD 3.5.0. Statistically significant differences are marked with an asterisk.

state to be collapsed. The former change may be well-motivated, since it was shown in Section 5.4.2 that conserving parameters can reduce overfitting and increase inference speed; however, it was also shown that providing the RNN component with more rather than less power is beneficial for achieving higher accuracy at the dependency parsing task. The latter change may reduce memory consumption during training (and inference to a lesser extent), but the network's inability to hide information makes it strictly less powerful. Table 5.4 compares systems trained with three different types of RNN cells: LSTMs, GRUs, and the *coupled input-forget gate* LSTM cells (Cif-LSTM) suggested by Greff et al. (2016), though the first tanh nonlinearity is no longer needed when using a coupled gate, so in this study it was removed.

While the LSTM and Cif-LSTM cells were fairly comparable, GRU cells performed quite abysmally compared to both LSTM variants. The reason for this may have to do with the high levels of dropout needed to achieve optimal performance. It seems likely that dropout encourages sparser output representations: if only 50% of nodes are kept after applying dropout, and the network uses sigmoid gates to manually set 50% of the nodes to near zero, then the system is only "surprised" by 25% of the zeros in the state. The idea that sparser output representations should be preferred to dense ones is supported by the generally accepted superiority of ReLU—which makes sparsity easy to achieve—over tanh. GRU cells, however, cannot effectively sparsify their states because their coupled input/forget gate forces each cell to update with either the current input or the previous cell but not neither, and the lack of output gate means that whichever value it takes is guaranteed to be revealed in the output state. By contrast, the output gate in the Cif-LSTM model allows it to maintain a sparse output state, which may help it adapt to the high levels of dropout needed to prevent overfitting. A manual examination of the LSTM gate biases learned by the fully trained vanilla model supports this dropout hypothesis. Most of these bias terms are nontrivially negative across all layers in all three gates, indicating that the network attempted to set the cell and output state to zero whenever possible. Table 5.4 also suggests that Cif-LSTM gates slightly underperform vanilla LSTM cells, but the difference is much smaller. Regarding speed, because the gate and candidate cell activations can be computed simultaneously with one matrix multiplication in the Cif-LSTM model, it turns out to be faster than the GRU version even though both cell types have the same number of parameters. This suggests that Cif-LSTM cells should be preferred to GRU cells when inference speed or parameter conservation is critical, being faster and more effective when trained

Model	UAS	LAS
Default	**95.75**	**94.22**
No word dropout	95.74	94.08*
No tag dropout	95.28*	93.60*
No tags	95.77	93.91*

Table 5.5: Ablation of embedding dropout. Test Accuracy on PTB-SD 3.5.0. Statistically significant differences are marked with an asterisk.

Model	UAS	LAS
$\beta_2 = .9$	**95.75**	**94.22**
$\beta_2 = .999$	95.53*	93.91*

Table 5.6: Comparison of optimizer hyperparameters. Test Accuracy on PTB-SD 3.5.0. Statistically significant differences are marked with an asterisk.

with dropout regularization.

Embedding Dropout

Because the parser has a lot of power, it also needs a lot of regularization. Thus the system includes embedding dropout, where whole words and tags are dropped as well as individual hidden nodes. Words and tags are dropped independently at a rate of 33% during training: when one is dropped the other is scaled by a factor of two to compensate, and when both are dropped together, the model simply gets an input of zeros. The labeled training dataset is relatively small (only about 40,000 sentences) and the system trains relatively fast (about 75 sents/sec), so a single pass through the entire training dataset takes under 10 minutes. Consequently, the system could see the same exact sequence more than a hundred times over the course of training. When whole embeddings are dropped 33% of the time, on the other hand, the probability of seeing a unique ten-word input sequence twice during the course of training is only about 0.1%. Thus embedding dropout (and dropout more generally) can be seen not only as a regularizer, but also as a quick-and-dirty means of data augmentation. Table 5.5 shows that models trained with only word or tag dropout but not both wind up signficantly overfitting, hindering label accuracy and—in the latter case—attachment accuracy. This suggests that preventing the system from seeing the same sequences repeatedly helps generalization accuracy. Interestingly, not using any tags at all actually results in better performance than using tags without dropout. While tags are still useful for neural parsing, it's clear that there needs to be some mechanism in place to keep the network from relying on them too much.

Optimizer

The system was optimized with Adam (Kingma and Ba, 2015), which (among other things) keeps a moving average of the L_2 norm of the gradient for each parameter throughout training and divides

		English PTB-SD 3.3.0		Chinese PTB 5.1	
Type	**Model**	**UAS**	**LAS**	**UAS**	**LAS**
Transition	Ballesteros et al. (2016)	93.56	91.42	87.65	86.21
	Andor et al. (2016)	94.61	92.79	–	–
	Kuncoro et al. (2016)	**95.8**	**94.6**	–	–
Graph	Kiperwasser and Goldberg (2016)	93.9	91.9	87.6	86.1
	Cheng et al. (2016)	94.10	91.49	88.1	85.7
	Hashimoto et al. (2017)	94.67	92.90	–	–
	Deep Biaffine	95.74	94.08	**89.30**	**88.23**

Table 5.7: Basic system results on the English PTB and Chinese PTB parsing benchmarks.

	Catalan		Chinese		Czech	
Model	**UAS**	**LAS**	**UAS**	**LAS**	**UAS**	**LAS**
Andor et al.	92.67	89.83	84.72	80.85	88.94	84.56
Deep Biaffine	**94.69**	**92.02**	**88.90**	**85.38**	**92.08**	**87.38**

	English		German		Spanish	
Model	**UAS**	**LAS**	**UAS**	**LAS**	**UAS**	**LAS**
Andor et al.	93.22	91.23	90.91	89.15	92.62	89.95
Deep Biaffine	**95.21**	**93.20**	**93.46**	**91.44**	**94.34**	**91.65**

Table 5.8: Basic system results on the CoNLL '09 shared task datasets.

the gradient for each parameter by this moving average, ensuring that the magnitude of the gradients will on average be close to one and that the average update will be close in magnitude to the learning rate. However, the value for β_2 recommended by Kingma and Ba—which controls the decay rate for this moving average—appears to be too high for this task (and we suspect more generally). When this value is very large, the magnitude of the current update is heavily influenced by the larger magnitude of gradients very far in the past, with the effect that the optimizer can't adapt quickly to recent changes in the model. Thus setting β_2 to .9 instead of .999 makes a large positive impact on final performance.

5.4.3 Results

This model gets nearly the same UAS performance on PTB-SD 3.3.0 as the current SOTA model from Kuncoro et al. (2016) in spite of its substantially simpler architecture, and gets SOTA UAS performance on CTB 5.1 as well as SOTA performance on all CoNLL 09 languages. It is worth noting that the CoNLL 09 datasets contain many non-projective dependencies, which require special augmentations for transition-based—but not arc-factored—parsers to predict. This may account for some of the large, consistent difference between this model and Andor et al.'s 2016 transition-based model applied to these datasets.

Where this model appears to lag behind the SOTA model is in LAS, indicating one of a few possibilities. Firstly, it may be the result of inefficiencies or errors in the GloVe embeddings or POS tagger, in which case using alternative pretrained embeddings or a more accurate tagger might improve label classification. Secondly, the SOTA model is specifically designed to capture phrasal compositionality; so another possibility is that this one doesn't capture this compositionality as effectively, and that this results in a worse label score. Finally, it may be the result of a more general limitation of arc-factored parsers, which have access to less explicit syntactic information than transition-based parsers when making decisions. Addressing these latter two limitations would require a more innovative architecture than the relatively simple one used in current neural arc-factored parsers.

5.5 Subsequent Work

5.5.1 Language transfer

The model proposed in Dozat and Manning (2017) has inspired or been extended by other researchers working on dependency parsing. Wang et al. (2017) aims to adapt the parser for low-resource languages with related high-resource languages, focusing on Singapore English (Singlish). They construct a small treebank of attested Singlish sentences with vocabulary and grammar very different from Standard American English. The crux of their approach involves "neural stacking" (Chen et al., 2016; Zhang and Weiss, 2016); they first train a deep biaffine parser on English Universal Dependencies (Nivre et al., 2016), and then they use its latent feature representations as additional input to another parser for Singlish with the same basic architecture (but different hyperparameters). First the English parser is run on the Singlish sentence, and the topmost recurrent state and each FFNN layer in the biaffine classifiers are extracted. Then the topmost recurrent state is concatenated to the Singlish word and tag embedding, and each FFNN representations of the pretrained English model is added to the parallel representation of the Singlish model. This allows a newly-trained Singlish parser to leverage knowledge of English when making decisions, and achieves quite dramatic improvement over just using a monolingual English or Singlish parser.

5.5.2 Transition-based parsing

Ma et al. (2018) take the biaffine approach to arc-factored parsing and work it into a neural transition-based parser. Instead of using FFNNs like Chen and Manning (2014); Andor et al. (2016) or Stack-LSTMs Kuncoro et al. (2016), they use a novel *stack-pointer* architecture. Their stack-pointer networks build on pointer networks (Vinyals et al., 2015), which can be thought of as sequence-to-sequence models where the input sequence is of token embeddings and the output sequence is of indices into the input sequence. It maintains a stack of words whose dependents have

not yet been partially but not fully parsed. The stack-pointer network, after being primed by the input sequence, "writes" to the output sequence all dependents of the word on top of the stack. When a word has multiple dependents, it is trained to parse its leftward dependents first, beginning with the closest one, then to parse its rightward dependents, starting with the closest one. Once all of a word's dependents have been fully parsed, the system points to the word itself, indicating that the word should be popped from the stack and that its parent should continue searching for more dependents. Critically, it uses a biaffine variable-class classifier like the one described in this chapter to identify a word's dependents. They compare this approach to their own reimplementation of the parser described here, finding that the two achieve comparable performance for English, Chinese, and German PTB datasets.

5.5.3 Constituency parsing

Teng and Zhang (2018) adapt the dependency parser for constituency parsing. They build two models, both of which use the biaffine transformations proposed in Dozat and Manning (2017) to generate scores. In one model (the *span* model), they use a biaffine scorer to compute the probability of a word i beginning a *span* (i.e. syntactic constituent) that another word j ends. In another model (the *rule* model), they use one biaffine scorer to compute the probability that a span between words i, j has left span i, k and another to compute the probability that it has right span $k + 1, j$, where $i \leq k < j$. Both are formulated so as to be decodable using the CKY parsing algorithm. They find that the biaffine components significantly improve over the affine alternatives. Their system achieves excellent performance on the PTB dataset, achieving comparable performance to the best published model that foregoes using ensembling or reranking.

5.5.4 Multitask dependency parsing/semantic role labeling

Shi and Zhang (2017) train a multitask system with two objectives: dependency parsing and semantic role labeling (SRL), focusing on the latter. Their approach involves training a shared sentence-encoding BiLSTM that feeds into smaller task-specific modules. The dependency parsing module includes one more BiLSTM layer, followed by the deep biaffine transformation described in Section 5.3. The SRL module likewise uses one more BiLSTM layer, but in addition to providing it with the topmost output state of the shared BiLSTM, they provide it with a lemma embedding and an *indicator* embedding, which indicates whether the word is a semantic predicate or not. They train their system with a weighted cross-entropy loss, finding that the joint training improves SRL significantly. Their multitask system achieves state-of-the-art performance on the CoNLL 2009 datasets for English, Chinese, and German (Hajič et al., 2009), which contain both dependency and SRL parses. It also reaches state-of-the-art performance on the English WSJ and Brown corpus SRL dataset without needing to use model ensembling or reranking.

5.5.5 Coreference resolution

Zhang et al. (2018a) build a coreference system that uses a biaffine variable-class classifier. They use a sigmoid classifier to classify each possible span of words as either being a mention or not (pruning away the most unlikely ones), and then use a biaffine scorer to classify each mention as being coreferent with a preceding one (or a special empty token). Their approach is relatively simple and intuitive but still consistently achieves state-of-the-art F_1. Critically, they find in an ablation study that replacing the biaffine classifier with a traditional FFNN one reduces performance down to just above the accuracy of the previous state-of-the-art system.

5.6 Conclusion

This chapter has focused on a dependency parser based on the biaffine classifiers motivated in Chapter 3.2. The deep biaffine parser was found to increase both speed and accuracy over alternatives that used feedforward classifiers or shallow biaffine classifiers. The larger but more heavily regularized network outperforms other neural arc-factored parsers and gets fairly comparable performance to the current state-of-the-art transition-based parser. This chapter has also provided empirical motivation for the proposed architecture and configuration over similar ones in the existing literature. Chapter 6 will explore ways to push this performance even higher, emphasizing languages with more productive morphology and orthography than English. Then Chapter 7 will extend the system to more challenging dependency schemes that relax the assumption that each token has a unique head.

Chapter 6

Multilingual Augmentations

6.1 Introduction

The previous chapter laid out a relatively simple neural dependency parser, and emphasized its performance on the standard English benchmark dataset, since the majority of competitive alternatives only evaluated on English. However, most languages aren't English, and present challenges that the basic, English-centric system is ill-equipped to handle. This chapter—originally published as Dozat et al. (2017) and Qi et al. (2018)—has a few goals, focusing primarily on adapting the system to languages that display a wider variety of linguistic phenomena. The biggest adaptation addresses the observation that many languages mark functional relationships with case inflections rather than with function words and word order. The system needs a way to identify these case inflections and what they contribute to the sentence; since inflections are generally recoverable from orthography, I provide the system with a character-level network that learns to construct word embeddings from the sequence of characters. In addition to building in a way of learning morphology from tokens, I explicitly incorporate the relative locations of each pair of tokens into the scoring function. This way, parsers trained on languages with more free word order can take advantage of the distance between two words when deciding how likely an edge is between them, and parsers trained on languages with more rigid word order can explicitly condition the score on linear order. Furthermore, I take advantage of the fact that variable-class classification is a kind of sequence labeling to train a deep affine (and subsequently deep biaffine) part-of-speech (POS) tagger analogous to the deep biaffine dependency parser that uses most of the same hyperparameters. This is to simultaneously ensure that an accurate POS tagger is available for the language being parsed (which may not be true for lower-resource languages) and to address the concern in the previous chapter that currently available taggers may not be accurate enough to maximize downstream parsing performance.

In order to fairly evaluate the efficacy of these adaptations, it's necessary to examine the system's performance on a wide variety of different languages and compare the results against a wide variety

of alternatives. To this end, the CoNLL shared task on Universal Dependency parsing (Nivre et al., 2016; Zeman et al., 2017; Nivre et al., 2017b,a) is an excellent testbed. Not only does it include training datasets for 45 different languages, but several dozen teams participated in it, providing many strong baselines. Many of these baselines use transition-based systems, an alternative paradigm to the arc-factored system proposed in this thesis. One advantage of the arc-factored approach over the transition-based one is that predicting crossing dependency edges is no more difficult than predicting non-crossing edges. Transition-based parsers, on the other hand, require more planning for these kinds of non-local constructions. Since crossing dependencies are common in languages with less restrictive word order, this raises the question—explored in this chapter—of whether arc-factored parsers are better at capturing crossing dependencies in such languages. Ultimately, the system achieved the highest performance on the 2017 shared task during the evaluation period according to all five relevant metrics: universal part-of-speech (UPOS), language-specific part-of-speech (XPOS), unlabeled attachment score (UAS), labeled attachment score (LAS), and content word labeled attachment score (CLAS).

One noteworthy feature of this approach is its relative simplicity. It uses a single tagger/parser pair per language, trained on only words and tags; there is no ensembling or language model pretraining, either of which could potentially push accuracy even higher. Because of its simplicity, accuracy, and speed, several independent teams extended it for the subsequent CoNLL 2018 shared task (Zeman et al., 2018).

6.2 Architecture

6.2.1 Deep biaffine parser

The basic architecture is the same as in Chapter 5, and will be reiterated here. In this parser, the input to the model is a T-length sequence of tokens and their part-of-speech tags $(w_t^{(\text{word})}, w_t^{(\text{pos})})$. Each word and tag in the sequence is given an embedding (the exact process will be described in Section 6.2.2), and for each token, its word and tag embeddings are concatenated. The sequence of embeddings is then put through a multilayer bidirectional LSTM network. The output state of the final LSTM layer is then fed through four separate ReLU layers, producing four specialized vector representations: one for the word as a dependent seeking its head; one for the word as a head seeking all its dependents; another for the word as a dependent deciding on its label; and a fourth for the word as head deciding on the labels of its dependents. These vectors are then used in two biaffine classifiers: the first computes a score for each pair of tokens, with the highest score for a given token indicating that token's most probable head; the second computes a score for each label for a given token/head pair, with the highest score representing the most probable label for the arc from the head to the dependent. This is shown graphically in Figure 6.1.

Put formally, given sentence i's sequence of T word embeddings $(\mathbf{x}_{i,1}^{(\text{word})}, \ldots, \mathbf{x}_{i,T}^{(\text{word})})$ and T

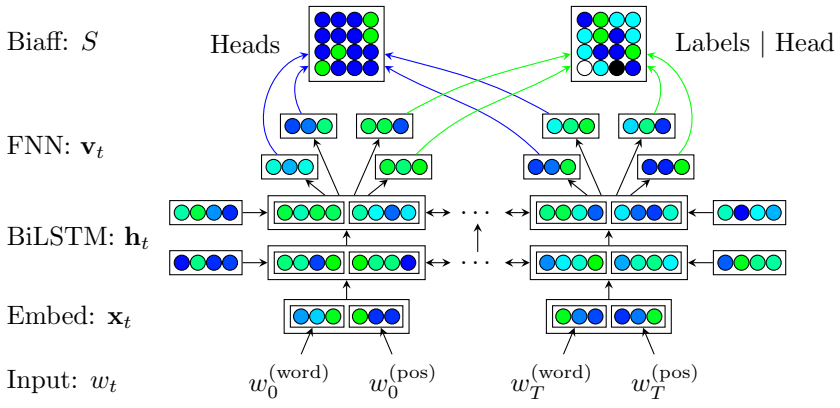


Figure 6.1: Basice parser architecture, repeated from Chapter 5. Word embeddings $\mathbf{x}_t$ are fed into a BiLSTM, which uses FNNs to get four separate representations $\mathbf{v}_t$ for each word. Two biaffine scorers turn these into score matrices S, where each cell represents the score of an edge or label between two tokens.

tag embeddings $(\mathbf{x}_{i,1}^{(\text{pos})}, \ldots, \mathbf{x}_{i,T}^{(\text{pos})})$, each pair is concatenated together and fed into a BiLSTM. As elsewhere in this thesis, lowercase italics will be for scalars, lowercase bold for vectors, uppercase italics for matrices, and uppercase bold for tensors, except where otherwise noted. The convention will be maintained when indexing and stacking; so $\mathbf{a}_i$ is the ith vector of matrix A, and matrix A is the stack of all vectors $\mathbf{a}_i$. Indices into function output will be notated before the function's arguments. To avoid notational clutter, the sentence index i will be dropped in following expressions.

$$\mathbf{x}_t = \mathbf{x}_t^{(\text{word})} \oplus \mathbf{x}_t^{(\text{pos})} \tag{6.1}$$

$$\mathbf{h}_t = \text{BiLSTM}_t\big(X\big)\big) \tag{6.2}$$

As described in Chapter 5, the system then produces four distinct vectors from each recurrent hidden state $\mathbf{h}_t$ using feedforward ReLU layers. The dependent representations are annotated without a tilde diacritic, and the head representations are annotated with. Similarly, vector pairs for the unlabeled arc (edge) classifier are annotated with a superscript (e), and vectors for the label classifier are annotated with (l).

$$\mathbf{v}_t^{(\text{e})} = \text{FNN}(\mathbf{h}_t) \qquad \text{Edge-dep} \tag{6.3}$$

$$\mathbf{v}_t^{(\text{l})} = \text{FNN}(\mathbf{h}_t) \qquad \text{Label-dep} \tag{6.4}$$

$$\tilde{\mathbf{v}}_t^{(\text{e})} = \text{FNN}(\mathbf{h}_t) \qquad \text{Edge-head} \tag{6.5}$$

$$\tilde{\mathbf{v}}_t^{(\text{l})} = \text{FNN}(\mathbf{h}_t) \qquad \text{Label-head} \tag{6.6}$$

As in the previous chapter, the edge score is a variable-class biaffine function (VCBiaff) of the edge

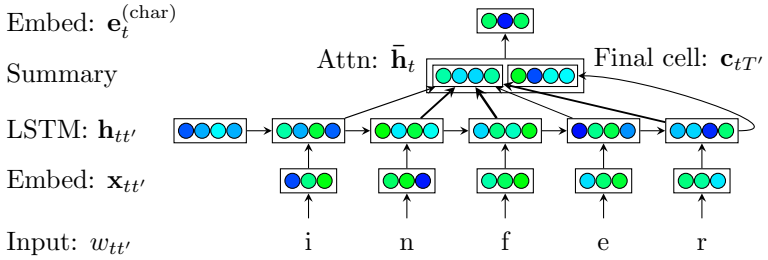


Figure 6.2: The architecture of the character-level embedding model. Character embeddings are fed into an LSTM, which is then summarized with attention and the final cell state. Finally, the concatenation of these is linearly transformed to the desired size.

representations (see Eq. 5.13), and the predicted head $\tilde{t}_t$ is the token that maximizes this score for word t.

$$\mathbf{s}_t^{(\text{e})} = \tilde{V}^{(\text{e})}\big(U\mathbf{v}_t^{(\text{e})} + \mathbf{w}\big) \tag{6.7}$$

$$= \text{VCBiaff}(\mathbf{v}_t^{(\text{e})}, \tilde{V}^{(\text{e})}) \tag{6.8}$$

$$\tilde{t}_t = \arg\max \mathbf{s}_t^{(\text{e})} \tag{6.9}$$

After deciding on a head $\tilde{t}_t$ for word t, a fixed-class biaffine transformation (FCBiaff) of the label-dep representation and the predicted head's label-head representation predicts the label:

$$\mathbf{s}_t^{(\text{l})} = \tilde{\mathbf{v}}_{\tilde{t}_t}^{\top(\text{r})}\mathbf{U}\mathbf{v}_t^{(\text{l})} + W\big(\mathbf{v}_t^{(\text{l})} \oplus \tilde{\mathbf{v}}_{\tilde{t}_t}^{(\text{l})}\big) + \mathbf{b} \tag{6.10}$$

$$= \text{FCBiaff}(\mathbf{v}_t^{(\text{l})}, \tilde{\mathbf{v}}_{\tilde{t}_t}^{(\text{l})}) \tag{6.11}$$

The biaffine approaches to classification in Eqs. (6.7, 6.10) have intuitive probabilistic and information-theoretic interpretations, described in detail in Chapter 3.

These two biaffine classifiers are optimized jointly by summing their softmax cross-entropy losses. At test time, a spanning tree algorithm that iteratively identifies and fixes cycles ensures that the tree is well-formed.[1]

6.2.2 Character-level model

The system in Chapter 5 represented words as the sum of a pretrained vector[2] and a holistic word embedding for frequent words. However, that approach seems insufficient for languages with rich

[1]The version used in the 2017 shared task uses a simpler version of the Chu-Liu/Edmonds algorithm (Chu and Liu, 1965; Edmonds, 1967) that doesn't guarantee a *maximum* spanning tree. The version for the 2018 shared task does, however.

[2]The pretrained vectors were provided for the shared task for most languages and were trained using word2vec (Mikolov et al., 2013); for Gothic, which had no provided vector embeddings, the embeddings came from Facebook's FastText vectors (Bojanowski et al., 2016).

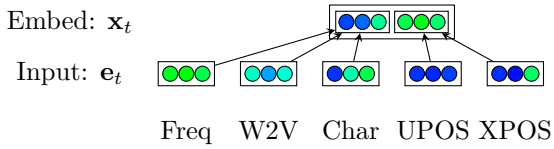


Figure 6.3: Total embedding architecture. The three diffierent kinds of word embeddings (frequent token, pretrained word2vec, and character-level) are added together. In the parser, the two different kinds of POS embeddings (UPOS, XPOS) are added together and concatenated with the word embeddings.

morphology. The system with only holistic embeddings will miss out on information about words that are absent or infrequent in the pre-training corpus but that have highly predictive morphological suffixes. To address this limitation, the system here generates an additional embedding composed from an LSTM over characters. Here, each character is given a trainable vector embedding, and the sequence of character embeddings is fed into a unidirectional LSTM. However, the LSTM produces a *sequence* of recurrent states, which must then be converted into a single vector. The simplest approach is to take the last one—which would represent a summary of all the information aggregated one character at a time—and linearly transform it to the desired dimensionality. Another approach, suggested by Cao and Rei (2016), is to use attention over the hidden states, and then transform the resulting context vector to the desired size. In theory, this should allow the model to learn morpheme information more easily by attending more closely to the LSTM output at morpheme boundaries. To get the best of both worlds, the system uses the recurrent output states for attention and the cell state for summarizing, shown in Figure 6.2. Formally, given a sequence of T' character embeddings for the t-th unique word in the minibatch, an LSTM computes a cell and hidden state for each character t' in the sequence. For notational simplicity, the word index t will be excluded from these equations, leaving only the character index t'.

$$\mathbf{x}_{t'} = \mathbf{x}_{t'}^{(\text{char})} \tag{6.12}$$

$$\mathbf{h}_{t'}, \mathbf{c}_{t'} = \text{LSTM\&Cell}_{t'}(X) \tag{6.13}$$

A linear attention mechanism then weights each hidden vector in H (Eqs. 6.14, 6.15). The weighted average of H is concatenated with the final cell state, and the two states together are linearly transformed to have the desired dimensionality (Eq. 6.16). Similar to the deep biaffine transformation proposed in Chapter 5, this allows for a large recurrent state but a smaller final embedding.

$$\mathbf{a} = \text{softmax}(H\mathbf{w}) \tag{6.14}$$

$$\bar{\mathbf{h}} = H^\top \mathbf{a} \tag{6.15}$$

$$\mathbf{e}^{(\text{char})} = W(\bar{\mathbf{h}} \oplus \mathbf{c}_{T'}) \tag{6.16}$$

In this way the hidden states are used for attention and the cell state is used as a holistic summary vector. The approach used here is admittedly somewhat arbitrary, and more empirical work is needed to determine the best way of generating word embeddings from a character-level LSTM for the dependency parsing task. After computing the character-level word embedding, the three types of word embeddings—word2vec (w2v), frequent token (freq), and character (char)—get added together element-wise. Similarly, the UPOS and XPOS tags are added together before being concatenated with the final word embedding. This is shown in Figure 6.3.

$$\mathbf{x}_t^{(\text{word})} = \mathbf{e}_t^{(\text{w2v})} + \mathbf{e}_t^{(\text{freq})} + \mathbf{e}_t^{(\text{char})} \tag{6.17}$$

$$\mathbf{x}_t^{(\text{pos})} = \mathbf{e}_t^{(\text{upos})} + \mathbf{e}_t^{(\text{xpos})} \tag{6.18}$$

The resulting two vectors are used as input to the BiLSTM parser in Eq. (6.1).

6.2.3 POS tagger

The final piece of this system is a separately-trained part of speech tagger. The architecture for the tagger is almost identical to that of the parser (and shares fundamental properties with other neural taggers; cf. Ling et al. (2015); Plank et al. (2016))—it uses a BiLSTM over word vectors (using the tripartite representation from Section 6.2.2), then uses ReLU layers to produce one vector representation for each type of tag.

Thus the system uses a BiLSTM over word embeddings, as with the parser architecture.

$$\mathbf{x}_t = \mathbf{x}_t^{(\text{word})} \tag{6.19}$$

$$\mathbf{h}_t = \text{BiLSTM}_t(X) \tag{6.20}$$

It then predicts each tag type (UPOS and XPOS, but not UFeats) using a different deep affine classifier. That is, it generates separate representations for each of UPOS and XPOS, and uses those as the input to affine classifiers. This makes the tagger maximally analogous to the dependency parser.

$$\mathbf{v}_t^{(\text{upos})} = \text{FNN}(\mathbf{h}_t) \tag{6.21}$$

$$\mathbf{v}_t^{(\text{xpox})} = \text{FNN}(\mathbf{h}_t) \tag{6.22}$$

$$\mathbf{s}_t^{(\text{u/xpos})} = \text{Aff}(\mathbf{v}_t^{(\text{u/xpos})}) \tag{6.23}$$

The tag classifiers are trained jointly using cross-entropy losses that are summed together during optimization, but the tagger is trained independently from the parser. Initial experiments involving jointly training the tagger and parser from one BiLSTM failed to produce results on par with the pipelined approach. These experiments included predicting tags, labels, and parses independently;

predicting tags from two lower layers of the BiLSTM and the labels and parses; and predicting tags first, then concatenating tag embeddings to the BiLSTM layers before making parse and label predictions. Even when unlabeled accuracy remained the same, labeled accuracy consistently suffered. There are two possible overlapping explanations for this. One is that the parser needs to incorporate POS tag information into the BiLSTM in order to accurately predict the label. The other is that error signal from the POS tagger during optimization overwhelms gradient from the labeler. Whatever the cause, the performance drop was too significant to justify using a single-network system over the two-network system used in this work.

6.3 Training details

This version of the system largely adopts the same hyperparameter configuration in the previous chapter, with a few exceptions. The parser uses three BiLSTM layers with 100-dimensional word and tag embeddings and 200-dimensional recurrent states (in each direction); the arc classifier uses 400-dimensional head/dependent vector states and the label classifier uses 100-dimensional ones; word and tag embeddings are dropped independently with 33% probability;[3] same-mask dropout (Gal and Ghahramani, 2016) is used in the LSTM, ReLU layers, and classifiers, with input and recurrent connections dropped with 33% probability; and the system is optimized with Adam (Kingma and Ba, 2015), at a learning rate of .002 and with $\beta_1 = \beta_2 = .9$. Models are trained for up to 30,000 training steps (where one step/iteration is a single minibatch with approximately 5,000 tokens), at first saving the model every 100 steps if fewer than 1,000 iterations have passed, and afterwards only saving if validation accuracy increases (or training accuracy for languages with no validation data). Training terminates when 5,000 training steps pass without improving accuracy.

The character model uses 100-dimensional uncased character embeddings with 400-dimensional recurrent states. Characters are not dropped, but similar to the core system the character model includes 33% dropout in the LSTM and attention connections.

The tagger uses nearly identical settings, except that the BiLSTM is only two layers deep, the dropout between recurrent connections is increased to 50%, and the character embeddings are case-sensitive.

There's substantial variability in training and testing speed across treebanks, but on an NVIDIA Titan X GPU the models train at 100 to 1000 sentences/sec and test at 1000 to 5000 sentences/sec. Even without GPU acceleration a tagger or parser can be run on an entire test treebank in ten to twenty seconds. By far the greatest runtime overhead comes not from the model itself, but from reading in the large matrices of pretrained embeddings, which can take several minutes. A full run over the 81 test sets on the TIRA virtual machine (Potthast et al., 2014) takes about 16 hours, but when parallelized on faster machines it can be done in under an hour.

[3] When only one is dropped, the other is scaled by a factor of two

	UPOS	XPOS	UAS	LAS	CLAS		UPOS	XPOS	UAS	LAS	CLAS
ar	89.36	**87.66**	76.59	71.97	68.17	*hsb*	90.30	99.84	67.83	60.01	**56.32**
ar_pud	71.17	0.00	58.87	49.50	46.06	*hu*	**95.34**	99.82	**82.35**	**77.56**	**76.08**
bg	**98.75**	**96.71**	**92.89**	**89.81**	**86.53**	*id*	**94.09**	99.99	85.17	**79.19**	**77.15**
bxr	84.12	99.35	**51.19**	30.00	25.37	*it*	**98.04**	**97.93**	**92.51**	**90.68**	**86.18**
ca	**98.59**	**98.58**	**92.88**	**90.70**	**86.70**	*it_pud*	**93.74**	2.48	**91.08**	**88.14**	**84.49**
cs	**98.83**	**95.86**	**92.62**	**90.17**	**88.44**	*ja*	88.14	89.68	75.42	74.72	65.90
cs_cac	**99.05**	**95.16**	**93.14**	**90.43**	**88.31**	*ja_pud*	89.41	7.50	78.64	77.92	68.95
cs_cltt	**97.91**	**89.98**	86.02	82.56	79.62	*kk*	57.36	55.72	43.51	25.13	19.32
cs_pud	96.42	**92.60**	**89.11**	**84.42**	**81.60**	*kmr*	90.04	89.84	47.71	35.05	28.72
cu	**95.90**	**96.20**	77.10	71.84	70.49	*ko*	**96.14**	**93.02**	**85.90**	**82.49**	**80.85**
da	**97.40**	99.69	**85.33**	**82.97**	**80.03**	*la*	**90.67**	**76.69**	**72.56**	**63.37**	**58.96**
de	**94.41**	**97.29**	**84.10**	**80.71**	**76.97**	*la_ittb*	**98.36**	**94.79**	**89.44**	**87.02**	**84.94**
de_pud	**85.71**	20.89	**80.88**	**74.86**	**73.96**	*la_proiel*	**96.72**	**96.93**	73.71	69.35	66.56
el	**97.74**	**97.76**	**89.73**	**87.38**	**83.59**	*lv*	**93.59**	**80.05**	**79.26**	**74.01**	**70.22**
en	**95.11**	**94.82**	**84.74**	**82.23**	**78.99**	*nl*	**93.24**	**90.61**	**85.17**	**80.48**	**75.19**
en_lines	**96.64**	**95.41**	**85.16**	**82.09**	**78.71**	*nl_lassysmall*	**98.39**	99.93	**89.56**	**87.71**	**85.22**
en_partut	**95.22**	**95.08**	86.10	82.54	77.40	*no_bokmaal*	**98.35**	99.75	**91.60**	**89.88**	**87.67**
en_pud	**95.40**	**94.29**	**88.22**	**85.51**	**82.63**	*no_nynorsk*	**98.11**	99.85	**90.75**	**88.81**	**86.41**
es	**96.59**	99.69	**90.01**	**87.29**	**82.08**	*pl*	**98.15**	**91.97**	**93.98**	**90.32**	**87.94**
es_ancora	**98.72**	**98.73**	**92.11**	**89.99**	**86.15**	*pt*	**97.24**	**83.04**	**89.90**	**87.65**	**83.27**
es_pud	88.39	1.76	**88.14**	**81.05**	**74.60**	*pt_br*	**98.22**	**98.22**	**92.76**	**91.36**	**87.48**
et	**93.01**	**95.05**	**78.08**	**71.65**	**69.85**	*pt_pud*	**88.99**	0.00	83.27	77.14	71.68
eu	**95.89**	99.96	**85.28**	**81.44**	**79.71**	*ro*	**97.59**	**96.98**	**90.43**	**85.92**	**81.87**
fa	**97.15**	**97.12**	**89.64**	**86.31**	**82.93**	*ru*	**96.99**	**96.73**	87.15	**83.65**	**81.80**
fi	**96.62**	**97.37**	**87.97**	**85.64**	**84.25**	*ru_pud*	86.85	**80.17**	**82.31**	**75.71**	**73.13**
fi_ftb	**96.30**	**95.31**	**89.24**	**86.81**	**84.12**	*ru_syntagrus*	**98.59**	99.57	**94.00**	**92.60**	**90.11**
fi_pud	**97.54**	0.00	**90.60**	**88.47**	**86.82**	*sk*	**96.87**	**85.00**	**89.58**	**86.04**	**83.86**
fr	96.20	98.87	**88.57**	**85.51**	**82.14**	*sl*	**98.63**	**94.74**	**93.34**	**91.51**	**88.98**
fr_partut	**96.16**	**95.88**	88.64	85.05	79.49	*sl_sst*	**94.04**	**86.87**	61.71	56.02	51.04
fr_pud	**89.32**	2.40	**83.45**	**78.81**	**77.37**	*sme*	86.81	88.98	51.13	37.21	39.22
fr_sequoia	**97.41**	99.06	88.48	86.53	83.37	*sv*	**97.70**	**96.40**	**88.50**	**85.87**	**83.71**
ga	**92.43**	**91.31**	78.50	**70.06**	**61.38**	*sv_lines*	**96.74**	**94.84**	**86.51**	**82.89**	**79.92**
gl	**97.72**	**97.50**	85.87	**83.23**	**78.05**	*sv_pud*	**94.33**	**92.33**	**81.90**	**78.49**	**76.48**
gl_treegal	**94.51**	**91.65**	78.28	73.39	66.02	*tr*	**93.86**	**93.11**	**69.62**	**62.79**	**60.01**
got	**95.74**	**96.49**	73.10	66.82	63.87	*tr_pud*	**72.73**	0.00	58.72	37.72	31.71
grc	**92.64**	**84.47**	**78.42**	**73.19**	**67.59**	*ug*	76.65	78.69	56.86	39.79	30.11
grc_proiel	**97.06**	**97.51**	78.30	74.25	68.83	*uk*	**94.31**	**79.42**	**81.44**	**75.33**	**71.72**
he	82.42	82.45	67.70	63.94	56.78	*ur*	**93.95**	**92.30**	**87.98**	**82.28**	**75.88**
hi	**97.50**	**97.01**	**94.70**	**91.59**	**87.92**	*vi*	75.28	73.56	46.14	42.13	38.59
hi_pud	**85.48**	**34.82**	**67.24**	**54.49**	**48.87**	*zh*	85.26	85.07	68.95	65.88	62.03
hr	**97.68**	99.93	**90.11**	**85.25**	**82.36**						

	UPOS	XPOS	UAS	LAS	CLAS
All treebanks	**93.09**	**82.27**	**81.30**	**76.30**	**72.57**
Large treebanks	**95.58**	**94.56**	**85.16**	**81.77**	**78.40**
Parallell treebanks	**88.25**	**30.66**	**80.17**	**73.73**	**69.88**
Small treebanks	**87.02**	**82.03**	70.19	61.02	54.76
Surprise treebanks	–	–	54.47	40.57	37.41

Table 6.1: Results on the CoNLL 2017 shared task. Includes each treebank in the shared task plus the macro average over all of them. State of the art performance by the system is in bold.

6.4 Results

This model uses the base tokenization and segmentation generated by an external system (UDPipe; Straka et al. 2016) and produces UPOS tags, XPOS tags, dependency arcs, and dependency labels. Thus the relevant metrics for the system are UPOS accuracy, XPOS accuracy, unlabeled attachment score, labeled attachment score, and content labeled attachment score. The results of the shared task are presented in Table 6.1; the system achieves the highest aggregated score on all five of these metrics. Following discussion explores where the model does particularly well, and where it can likely be improved. Evaluations are done on CLAS performance, which is a principled extension of the common practice of removing punctuation from evalution. The shared task included four surprise languages, which lacked training data and thus required some sort of transfer learning approach.

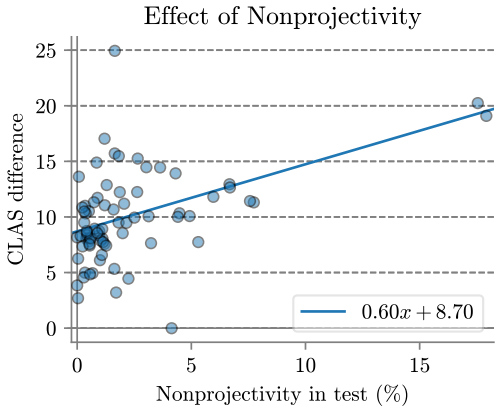
Because of the substantially different training and testing conditions, surprise languages will be excluded from the following analyses.

One small point to that end is that arc-factored systems such as this require tokenization and segmentation to have already been done. Jointly training a segmenter and arc-factored parser would be difficult, either requiring a reinforcement learning objective or consuming a large amount of memory quadratic in the number of characters in each paragraph or document. The system was therefore trained on gold segmentation and evaluated using the segmentation provided by UDPipe (maintained in the following experiments). For most treebanks this was easily sufficient, but for Vietnamese, Chinese, Japanese, and Arabic, UDPipe's lower performance at segmenting or tokenizing was correlated with a relatively large gap between CLAS and gold-aligned CLAS. Because the model achieves comparable CLAS and gold-aligned CLAS for nearly all other treebanks, this probably indicates that alignment errors propagated through the system into parsing errors.

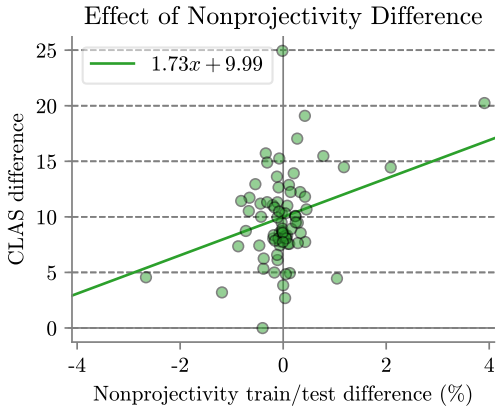
6.4.1 Nonprojectivity

In Universal Dependencies, unlike many other popular benchmarks, several treebanks have a relatively large number of crossing dependencies. This means that any competitive system will need to be able to produce non-projective arcs. One of the most frequently used approaches for producing fully non-projective parsers in transition-based systems is to add the `swap` action (Nivre, 2009). This makes any arbitrary non-projective arc possible, but increases the number of transition steps and the degree of planning required to produce that arc. One potential concern is that it might be more difficult to produce these longer chains of transition sequences, which would bias the model toward overproducing projective arcs. In an arc-factored system, by contrast, there's little reason to think non-projective arcs should be harder to predict than projective ones. One might hypothesize that because of this, a transition-based `swap`ping system would need to see more examples of crossing dependencies than a arc-factored system in order to generalize well. Thus this first study aims to explore how the fraction of non-projective arcs in a treebank affects the performance of the two types of systems.

To test the relative performance of an arc-factored and a transition-based model, the difference in per-treebank CLAS performance between this parser and the transition-based UDPipe v1.1 baseline (Straka et al., 2016) is plotted against the frequency of non-projective arcs in the test set in Figure 6.4a. To determine whether there is a significant relationship between the difference in performance, the data is fit to a generalized linear mixed effects regression model (Fisher, 1930), using Markov chain Monte Carlo sampling (Hadfield, 2010). Log data size, morphological complexity (see Section 6.5.2), and training set projectivity are included as random effects. The learned regression lines are plotted alongside the data. The difference between the performance of the arc-factored and transition-based parsers increases with the nonprojectivity of the test set significantly



(a) By absolute percentage of crossing dependencies in the test set. Difference in CLAS between this system and the transition-based UDPipe v1.1 as a function of the nonprojectivity of the test set.



(b) By relative percentage of crossing dependencies in the test set. Difference in CLAS between this system and UDPipe v1.1 as a function of the difference between the nonprojectivity of the test and training sets.

Figure 6.4: Comparison of parsing paradigm on crossing edges. How the percentage of non-projective arcs in the training and test set influence the accuracy of the proposed arc-factored and a transition-based parser. The transition-based sysem underperforms more in the presence of more crossing edges.

($p < 0.001$). This remains significant even when outliers[4] are excluded ($p < 0.05$). To the extent that UDPipe represents a typical non-projective transition-based parser, this result suggests that an arc-factored approach is better suited to parsing UD treebanks that have significant syntactic freedom or complexity than a transition-based one.

The data shown in Figure 6.4b further support the hypothesis that arc-factored parsers are better at handling nonprojectivity than transition-based ones. Figure 6.4b shows the difference between the projectivity of each test and training set, with regression lines generated analogously to Figure 6.4a (with data size, morphological complexity, and train/test nonprojectivity as random effects). When the training set has drastically fewer crossing dependencies than the test set, the arc-factored model achieves relatively higher accuracy; but when the transition-based parser can train on many crossing arcs, the models are closer in performance ($p < 0.001$), even when excluding the same outliers ($p < 0.05$). This suggests that the arc-factored approach learns and generalizes crossing dependencies more efficiently than the transition-based approach, although again this comes with the assumption that UDPipe's parser is representative of most transition-based `swap`ping parsers when it comes to producing nonprojective parses.

One important caveat is that the reported results for UDPipe v1.1 for each treebank use the transition system that maximized the validation accuracy for that language, which may have included transition systems without the `swap` action. This means that for languages with a relatively small fraction of nonprojective edges, the `swap`ping system may have achieved lower performance than what is reported here. Consequently, whenever UDPipe chose not to use `swap`, it tacitly admitted that `swap` isn't a one-size-fits-all solution to handling different proportions of crossing dependencies.

6.4.2 Data size

All trained parsers here use the same hyperparameter configuration, regardless of how much training data there is for the treebank each one was trained on. This means some systems may have overfit to small training datasets or underfit to large ones. One way to test this hypothesis is to compare performance of the system presented here to the highest-performing alternative system for each treebank in the shared task. Thus Figure 6.5 shows the per-treebank difference between the test CLAS performance of the model proposed in this work and that of the highest-performing model other than this one, plotted against the log training data size. On average, this system tends to do relatively better on larger datasets compared to other approaches and worse on smaller ones, according to a mixed effects regression model with train/test projectivity and morphological complexity set as random effects ($p < 0.001$). When the outliers are excluded,[5] this tendency is still significant ($p < 0.001$). This suggests that the model is overfitting to smaller datasets, and that increasing regularization or decreasing model capacity may improve accuracy for lower-resource languages.

[4] Korean (top); Ancient Greek, Latin (right)

[5] Kazakh, Uyghur (left); Japanese (bottom); Czech-CAC, Russian-SynTagRus, Czech (right)

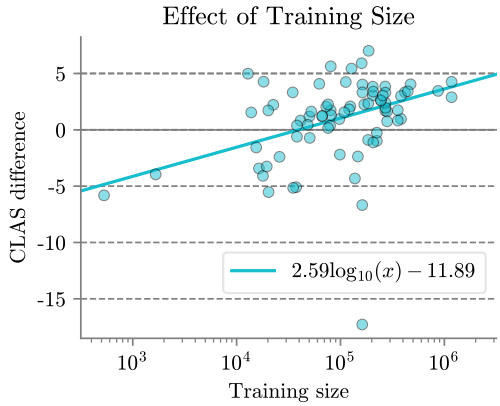


Figure 6.5: Evaluation of treebank size. Performance difference between this model and the highest-performing model other than ours as a function of log training data size. It tends to do better on larger datasets.

6.5 Ablation Studies

6.5.1 POS Tagger

This system uses its own tagger instead of provided ones; here, the goal is to motivate this choice with extrinsic tagger evaluation. That is, the question under investigation is "does using a more accurate tagger significantly increase parser accuracy?" This can be ascertained by comparing the performance of a system trained with the tagger presented here to an identical system trained on the baseline tags provided by UDPipe v1.1. Figure 6.6 plots the difference in accuracy between these parsers against the difference in accuracy between the taggers for each treebank. One can observe two things from this. The first is that the performance difference between the set of models trained on the deep affine tagger is statistically significantly better than the performance of the models trained on UDPipe tags, according to a Wilcoxon signed-rank test ($p < 0.001$). The second is that this can be explained by the improvement of the deep affine tagger over UDPipe v1.1, again accounting for dataset size, nonprojectivity, and morphology in a mixed effects model ($p < 0.001$). That is, as the performance of the tagger gets higher and higher, so does the accuracy of the parser trained on its output. This suggests that improving upstream tagger performance is an effective way of improving downstream parser accuracy. One small final note is that there was no significant correlation between the effect of training size on the difference in system/UDPipe-tagged parser performance ($p > 0.05$). That is, one might think that high-quality tags are only useful for smaller datasets, whereas larger datasets can compensate for low-quality tags through sheer volume of training data. However, there is no evidence here that this is the case.

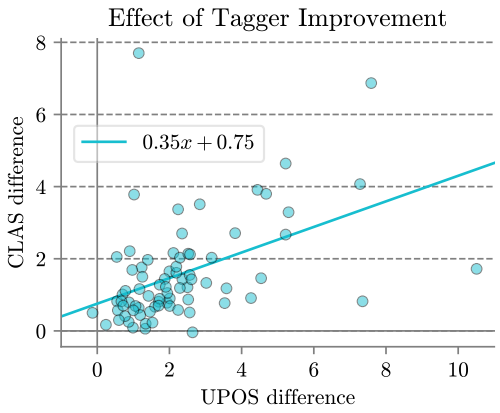


Figure 6.6: Comparison of relative tagger accuracy on parser performance. CLAS performance difference between a version of the system trained on its predicted tags and a version trained on UDPipe v1.1 tags as a function of the performance difference between the system's tagger and UDPipe's tagger. Using the proposed tagger improves performance.

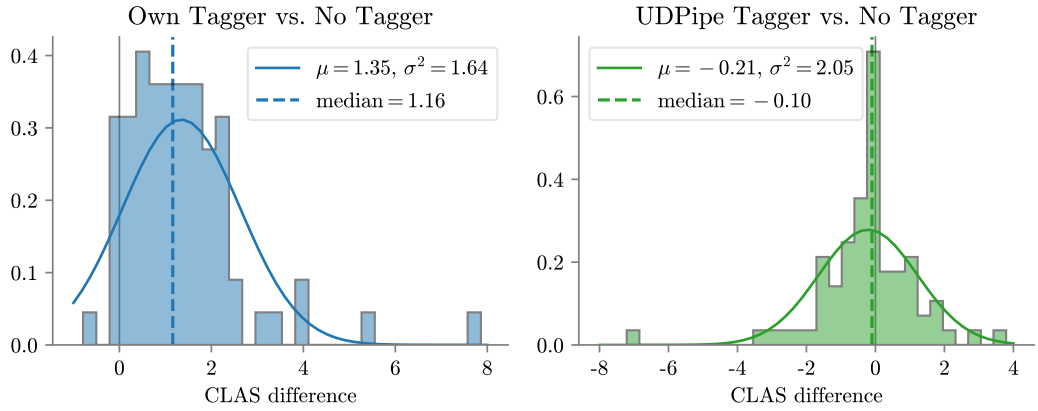


Figure 6.7: Comparison of tagger choice on parser performance. Performance difference between parsers using the system tagger and parsers without tags (left) and between parsers using UDPipe v1.1's tags and parsers without tags (right), with both histograms fit to normal distributions. Only using the proposed tagger improves performance over not using any tags.

The parsing approach laid out in this chapter uses one neural network to tag the sequences of tokens, and a second neural network to produce a parse from the tokens and tags. One might ask to what extent the tagger network is actually necessary, for a number of reasons: presumably whatever predictive patterns it learns from the token sequences would also be learnable by the parser network; errors by the tagger are likely to be propagated by the parser; and Ballesteros et al. (2015) found that POS tags are drastically less important for character-based parsers. If somewhat accurate tags always help, then a system without any tags should underperform a system trained with only UDPipe tags (Notag < UDPipe < DeepAff). If only highly accurate tags improve performance, then the base system should outperform the no-tag system but the UDPipe-tagged system should perform no different or worse (UDPipe ≤ Notag < DeepAff). Finally, if tags are no longer useful for modern neural parsers, but less accurate tags can propagate errors, then the UDPipe-tagged system should be worse than the other two, with no difference between the no-tag parser and base parser (UDPipe < Notag = DeepAff). Figure 6.7 makes this comparison, showing the difference between the base system and a variant trained without tags as well as the difference between a system trained with baseline UDPipe tags and a variant without them. It can be seen that the variant with no POS tag input is likewise significantly worse than the proposed model according to a Wilcoxon signed-rank test ($p < 0.001$), but not statistically different from the one trained with UDPipe tags ($p > 0.05$). This suggests that predicted POS tags are still useful for achieving maximal parsing accuracy in this system, provided the tagger's performance is high enough, and that the UDPipe baseline POS tagger isn't accurate enough for downstream tasks.

6.5.2 Character model

Many languages express grammatical relationships with inflectional morphology. Orthographically, these inflections are normally written as part of the (whitespace-separated) word they inflect. This creates problems for systems that represent words holistically; whole-token embeddings have no built-in mechanism for giving similar representations to orthographically similar words, so the system will be unable to identify any highly predictive case ending or infectional suffix in a rare or unseen token. Inflectional morphology often takes the place of word order or function words, which systems trained with only whole-token embeddings rely on. As such, any competitive system will need a way of addressing this limitation of token embeddings. Character-based embedding models, such as the one proposed here (which is similar to that of Ballesteros et al. (2015)), are designed to learn relationships between words with similar orthography. By hypothesis, it should allow the model to more effectively learn the relationships between words in languages with productive orthographic conventions and loose word order without an abundance of function words.

This hypothesis can be tested using another ablation study. As the inflectional complexity of UD treebanks increases, so too should the difference in performance between a model with a character-level word embedding compared to a model without. In this study, a second set of taggers and

parsers was trained on the dataset with only whole token and pretrained vectors, leaving out the vector composed from character sequences (for maximal comparability, the token-based parsers were trained with the original character-based taggers).

Of course, investigating this requires having some means of approximating the inflectional complexity of a language. Quantifying inflectional complexity directly for a given language at minimum requires text with not just morphological features, but morphological *segmentation*. Since Universal Dependencies lacks morphological segmentation, inflectional complexity can only be approximated. A reasonable approach to approximate inflectional complexity involves Herdan's law (Eq. 6.24; Herdan 1960; also known as Heaps' law, Heaps 1978), which more or less follows from Zipf's law (Zipf, 1949; van Leijenhorst and Van der Weide, 2005). Herdan's law observes that as one reads through a corpus of text, the log number of unique tokens one has read will be an affine function of the log total tokens (including repeats) that one has read.

$$\ln(|V|) = \beta \ln(|X|) + \alpha \tag{6.24}$$

Intuitively, in an inflectionally complex language, the ratio between the size of the vocabulary $|V^{(X)}|$ of a corpus to the size of the corpus itelf $|X|$ will be relatively high, because the same lemma may occur with many different contexts with many different forms; but in an inflectionally simplex language, that ratio will be smaller for the same size corpus, because a given lemma will normally appear with only a few forms. Put another way, given a corpus X of an inflectionally rich language, the addition of a word randomly drawn according to a Zipf distribution has a relatively high probability of being a lemma seen in X but with a new morphological form. Adding this word to X would increase both the size of the corpus $|X|$ and the size of the vocabulary $|V|$. In an inflectionally poor language, a randomly choosen word with a lemma already present in X is unlikely to have a new morphological form not seen in X, meaning the corpus size $|X|$ will increase but the vocabulary size $|V|$ won't. Therefore, one would expect the parameter β of Equation 6.24 to be higher for languages with rich morphology. Of course, it should be noted other kinds of morphological or orthographic complexity could also be responsible for a large vocabulary-to-corpus ratio, such as the kind of productive compounding found in German. Thus this approach is using *orthographic productivity* as an approximation for *inflectional complexity*.

Computing this β value for each treebank, the results are generally intuitive (although not without some variation, attributable to domain and dataset size). Somewhat surprisingly, Hindi and Urdu, which have some fusional morphology, are among the lowest, having $\beta = .555$ and .585 respectively, though this could be attributed to them coming from an unusually restricted domain. English and Vietnamese, fairly analytic languages, are also relatively low, with .631 and .661. Spanish and Portuguese, which have fusional morphology but still rely heavily on function words, have .7 and .704. The highly agglutinative Finnish, Estonian, and Hungarian have some of the highest, at .806, .822, and .846. Thus it is reasonable to conclude that the coefficient β in Equation 6.24 can

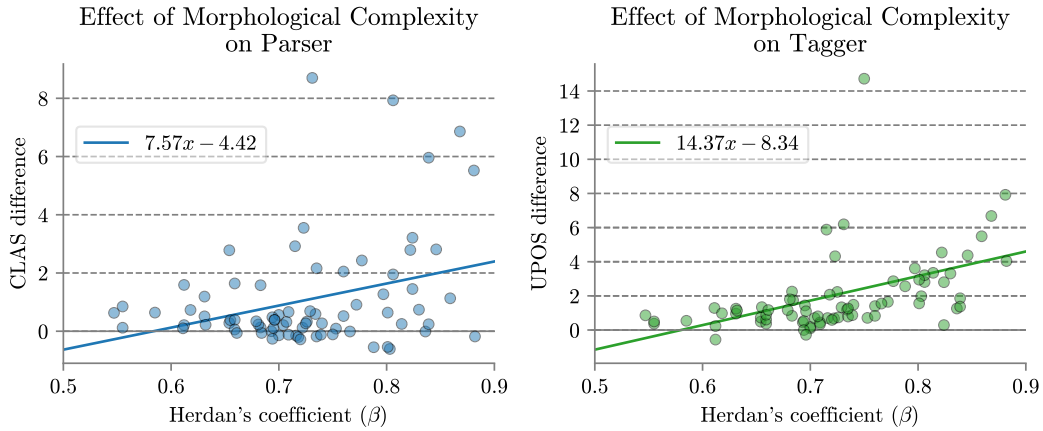


Figure 6.8: Character-level model ablation. Performance difference between the character-based approach and a pure token-based approach for parsing (left) and tagging (right) as a function of approximate morphological productivity. More productive languages see more improvement from the character model.

be used as a good approximation for inflectional richness.

Figure 6.8 plots the difference between models trained with and without character-level word embeddings against this value in Figure 6.8. According to a Wilcoxon signed rank test, the difference between the two approaches is statistically significant for the taggers ($p < 0.001$) and parsers ($p < 0.001$). Figure 6.8 also includes the regression line for a mixed effects model of the data with treebank size and training/test projectivity as random effects. Again, the character-level approach tends to significantly improve performance more as orthographic productivity grows both for parsing ($p < 0.005$) and tagging ($p < 0.001$). This suggests that incorporating subword information into UD parsing models is a promising way to improve performance on languages with rich inflection systems.

6.6 2018 Shared Task Extensions

The previous sections laid out extensions to the deep biaffine dependency parser discussed in Chapter 5 and the impact that they had on the 2017 CoNLL shared task. The first extension—a deep affine POS tagger—was designed to improve label accuracy, which is one place where the original version underperformed. The second extension—a word embedding composed from character sequences—was designed to improve accuracy on rare words in languages with heavy inflectional morphology. This section will describe several more extensions, including those used in Stanford's submission to the 2018 CoNLL Shared Task on UD parsing (Zeman et al., 2018): forcing consistency between the different kinds of tags, and building into the parser the explicit ability to capture distance and

linearization between a dependent and its possible heads.

6.6.1 Biaffine tagger

One evaluation metric not discussed previously is the "AllTags" metric, which measures what fraction of tokens were perfectly tagged according to their Universal POS tag (UPOS), their language-specific POS tag (XPOS), and their Universal Features (UFeats). For now, the discussion will focus on UPOS and XPOS, bringing in UFeats after an initial solution is found. The original tagger used a naïve approach to multitask tagging: use one deep affine classifier to predict the UPOS tags, and one more deep affine classifier to predict the XPOS tags. However, in this naïve approach, both modules makes errors independently (Eq. 6.28; note that the (u/x) superscript means that the equation applies for both UPOS or XPOS tags).

$$\mathbf{v}_t^{(\mathrm{u/x})} = \mathrm{FNN}(\mathbf{h}_t) \tag{6.25}$$

$$\mathbf{s}_t^{(\mathrm{u/x})} = \mathrm{Aff}\left(\mathbf{v}_t^{(\mathrm{u/x})}\right) \tag{6.26}$$

$$P\left(c_k^{(\mathrm{u/x})}|\mathbf{f}\right) = \mathrm{softmax}_k(\mathbf{s}_t^{(\mathrm{u/x})}) \tag{6.27}$$

$$P\left(c_k^{(\mathrm{u})}, c_{k'}^{(\mathrm{x})}|\mathbf{f}\right) = P\left(c_k^{(\mathrm{u})}|\mathbf{f}\right) P\left(c_{k'}^{(\mathrm{x})}|\mathbf{f}\right) \tag{6.28}$$

The consequence of Eq. (6.28) is that if one tagging module (say, UPOS) has an accuracy of 95% and the other (say, XPOS) has an accuracy of 90%, then the two modules together will have an expected accuracy of 90%% = 85.5%. This is a hardly trivial 4.5% lower than the highest possible accuracy that the joint probability model can achieve, which occurs if the higher-accuracy module only makes mistakes on the same tokens that the lower-accuracy modules makes them (90% in this example). Furthermore, the lower bound occurs when no errors overlap, and in this example is 85.0%, a mere half a percent lower than the expected accuracy. These inconsistent tags could confuse downstream systems, which might not know which of the annotations to trust. Of course, in practice errors aren't going to be independent, since some sentences or constructions are *a priori* more confusing than others. But this example highlights the dangers of falsely assuming independence.

To empirically motivate this problem, I trained taggers that assume independence between UPOS and XPOS on the 36 treebanks in the CoNLL 2018 shared task with official validation data and fewer than 250 unique XPOS tags. On average, in 2.70% of tokens (macro-averaged across treebanks), the system got one of the UPOS or XPOS tag right, and the other one wrong. This suggests that there is room for improvement for the multitask tagger.

The ideal classifiers for this task can be derived from the joint probability, where errors are by definition not independent. The joint conditional probability of a UPOS/XPOS tag pair can be

rewritten into the product of two dependent conditional probabilities.

$$P(c_k^{(\mathrm{u})}, c_{k'}^{(\mathrm{x})}|\mathbf{f}) = \frac{P(c_k^{(\mathrm{u})}, c_{k'}^{(\mathrm{x})}, \mathbf{f})}{P(\mathbf{f})} \tag{6.29}$$

$$= \frac{P(c_k^{(\mathrm{u})}, c_{k'}^{(\mathrm{x})}, \mathbf{f})}{P(\mathbf{f})} \frac{P(c_k^{(\mathrm{u})}, \mathbf{f})}{P(c_k^{(\mathrm{u})}, \mathbf{f})} \tag{6.30}$$

$$= \frac{P(c_k^{(\mathrm{u})}, c_{k'}^{(\mathrm{x})}, \mathbf{f})}{P(c_k^{(\mathrm{u})}, \mathbf{f})} \frac{P(c_k^{(\mathrm{u})}, \mathbf{f})}{P(\mathbf{f})} \tag{6.31}$$

$$= P(c_{k'}^{(\mathrm{x})}|c_k^{(\mathrm{u})}, \mathbf{f}) P(c_k^{(\mathrm{u})}|\mathbf{f}) \tag{6.32}$$

Eq. (6.32) reduces to Eq. (6.28) if $c_k^{(\mathrm{u})}$ and $c_k'^{(\mathrm{x})}$ are assumed to be conditionally independent. $P(c_k^{(\mathrm{u})}|\mathbf{f})$ is the same in both equations, so computing it with a deep affine classifier is still sensible. $P(c_{k'}^{(\mathrm{x})}|c_k^{(\mathrm{u})}, \mathbf{f})$ has changed, however. This term represents the probability of a particular XPOS class given not only the neural features $\mathbf{f}$, but also given the UPOS class $c_k^{(\mathrm{u})}$. Chapters 3.2 and 5 discuss cases similar to this extensively; it should come as little surprise then that this probability reduces to a biaffine softmax classifier. In these equations, the input features $\mathbf{f}$ are—as usual—a function of the BiLSTM. The predicted UPOS class $c_k^{(\mathrm{u})}$ will need its own features $\mathbf{f}\tilde{\mathbf{f}}$, which will actually be fixed, independent of the BiLSTM. When making a dependency label classification in the biaffine parser, the label is conditioned on one of T_i possible heads, which change from example to example and don't have a fixed representation. This precludes using a fixed embedding matrix to represent the possible classes (in that case, head words). On the other hand, when making an XPOS classification in the biaffine tagger, the tag is conditioned on one of m classes that *don't* vary by example. Consequently, the class features $\tilde{F}$ *can* have a fixed embedding representation $E^{(\mathrm{u})}$, usable in a biaffine classifier.

$$\mathbf{s}_t^{(\mathrm{u})} = \mathrm{DeepAff}(\mathbf{h}_t) \tag{6.33}$$

$$P(c_k^{(\mathrm{u})}|\mathbf{f}) = \mathrm{softmax}_k(\mathbf{s}_t^{(\mathrm{u})}) \tag{6.34}$$

$$\tilde{k}_t = \arg\max \mathbf{s}_t^{(\mathrm{u})} \tag{6.35}$$

$$\mathbf{s}_t^{(\mathrm{x})} = \mathrm{Biaff}(\mathbf{v}_t^{(\mathrm{x})}, \mathbf{e}_{\tilde{k}_t}^{(\mathrm{u})}) \tag{6.36}$$

$$P(c_{k'}^{(\mathrm{x})}|c_k^{(\mathrm{u})}, \mathbf{f}, \tilde{F} = E^{(\mathrm{u})}) = \mathrm{softmax}_k(\mathbf{s}_t^{(\mathrm{x})}) \tag{6.37}$$

Eqs. (6.33–6.35) make a UPOS prediction, and Eqs. (6.36, 6.37) use the predicted UPOS tag to inform the XPOS prediction. The XPOS tag is now conditionally dependent on the UPOS tag; the system will have access to more information when making XPOS predictions. This means that when the UPOS tag is right, the system should have a higher chance of getting the XPOS tag right than if it had no additional information, but when the UPOS tag is wrong, the error is more likely

to propagate. This is actually the desired behavior—UPOS and XPOS tags should be correct or incorrect together more often, which will increase the AllFeats accuracy; and even when the UPOS and XPOS tags are both wrong, they're at least more likely to be internally *consistent* with each other.

The UFeats tags can be done similarly. In the version of the system used in the CoNLL 2018 shared task Qi et al. (2018), the classifier for each universal feature is conditioned on a dedicated *UFeats* vector $\mathbf{v}_{it}^{(\mathrm{f})}$ similar to the UPOS and XPOS representations $\mathbf{v}_{it}^{(\mathrm{u/x})}$, as well as the UPOS embedding of the gold or predicted tag. There are other ways this could be done, such as by conditioning it on an XPOS embedding instead, or by adding or concatenating the UPOS and XPOS embeddings and conditioning on the result.

One issue that arises from this approach is that when the XPOS tagset (which varies by language) is large—perhaps because the tags are compositional, with each character representing a distinct morphological feature of the token—the bilinear tensor $\mathbf{U}$ can be prohibitively large. Computing the gradient normally means constructing multiple tensors of shape $(b \times d \times m)$, where b is the number of tokens per batch, m is the number of tags, and d is the hidden sizes of the two feature vectors. When using a tagset of about $m = 50$ (as in the UD dependency labels) a training batch size of $b = 2000$ tokens, hidden sizes of $d = 400$, and 32-bit floats, this can be computed with a perfectly manageable 120MB of GPU memory. However, expanding the tagset up to $m = 1000$ (which can happen for composite tagsets) increases memory requirements by the same factor, resulting in a 2.4GB memory footprint for each $(b \times d \times m)$, which can result in resource allocation errors. One way around this is to enforce that each slice of $\mathbf{U}$ be diagonal. Then the bilinear component reduces to Eq. (6.38), which is much more computationally feasible.

$$\tilde{\mathbf{x}}_i \mathbf{U} \mathbf{x}_i = U(\mathbf{x} \odot \tilde{\mathbf{x}}_i) \tag{6.38}$$

Computing the gradient can be done with tensors of size $(b \times m)$, and $(b \times d)$, which require much less memory. Thus a diagonal $\mathbf{U}$ tensor can be used to conserve parameters when necessary, although it was not explored for the 2018 shared task.

This biaffine approach to XPOS tagging and UFeats tagging was motivated by the desire to improve consistency between the tags. In fact, an actual comparison of the deep biaffine approach to a baseline system paints an even more favorable picture. The baseline system aims to improve consistency by sharing the ReLU layer in the deep affine classifers across the three types of tags. In theory, the three tagsets will be forced to use the same features, encouraging them to fit the data along similar patterns. Figure 6.9 shows the accuracy of the differing tagging metrics across the datasets in the CoNLL 2018 shared task for the biaffine and shared layer approaches. The UPOS accuracy is comparable across the two conditions, which is to be expected because both variants use a deep affine classifier for the UPOS tags. This suggests that restricting the representational capacity to encourage consistency in the shared layer condition didn't hurt performance over the original

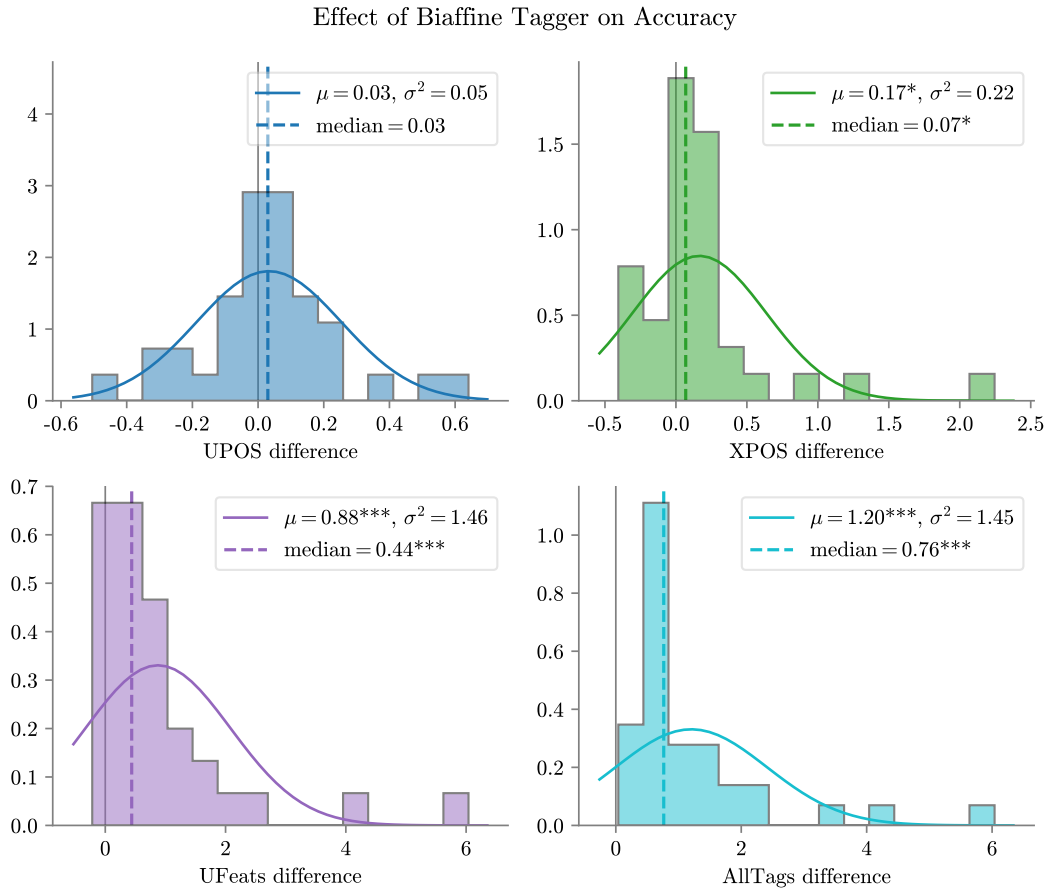
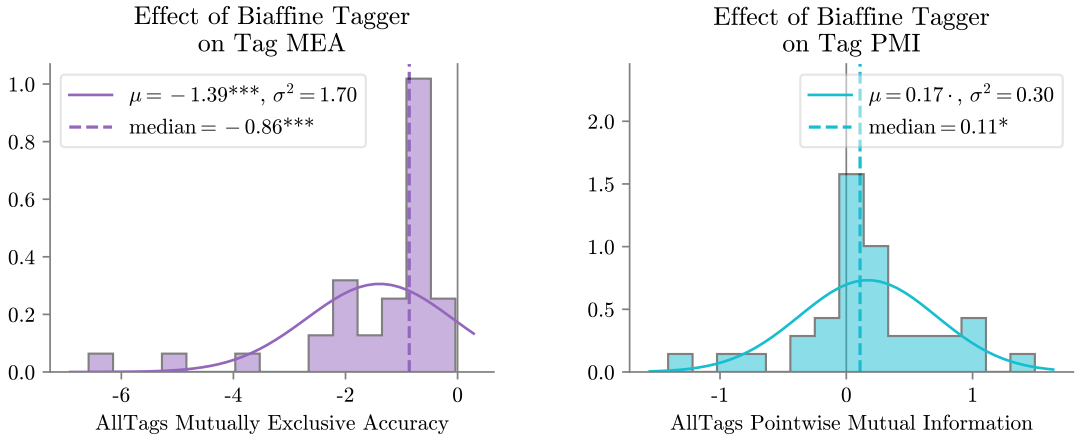


Figure 6.9: Tagger biaffinity ablation, evaluating on accuracy. Difference in raw accuracy between the biaffine and shared hidden approaches to maintaining consistency. Statistical significances are marked with an asterisk. The biaffine approach improves XPOS and UFeats accuracy.



(a) Difference in mutually exclusive accuracy (getting some but not all tags correct) between the shared and biaffine approaches (lower is better).

(b) Difference in pointwise mutual information (how non-independent the three accuracy measures are) between the shared and biaffine approaches (higher is better).

Figure 6.10: Tagger biaffinity ablation, evaluating on consistency.
Difference in consistency between the biaffine and shared hidden approaches. The biaffine approach appears to improve consistency by both measures.

deep affine tagger with separate hidden layers. Unexpectedly, XPOS accuracy actually improved globally when conditioned on UPOS predictions. Rather than making more mistakes when the UPOS module made mistakes and making fewer mistakes when the UPOS module was correct, the XPOS module actually made fewer mistakes overall. This effect is even more dramatic in the UFeats module. While the UPOS difference is not statistically significant according to a paired t-test (which tests the average difference) or a paired rank-sum test (which tests the median), both other modules are significant by both tests (($p < .05$) for XPOS and ($p < .001$) for UFeats, respectively). This substantial improvement impacts the AllFeats metric as well ($p < .001$). In the four histograms, there are three clear outliers. These represent Ancient Greek (Perseus), Czech (PDT), and Czech (CAC), which all have very large tagsets where each character represents a morphological feature. This suggests that the UPOS prediction significantly prunes the possible XPOS label space and helps to identify which morphological features are possible for the token under consideration. This may help the system identify morphological patterns among different parts-of-speech, such as that adjectives and nouns but not verbs are defined for case and gender.

By hypothesis, the consistency of the biaffine approach should be superior to the consistency of the shared layer. There are at least two ways that consistency can be measured. The first is by looking at what might be called the *mutually exclusive accuracy* (MEA), the fraction of times that the system gets some tags right but not all of them. However, the mutually exclusive accuracy

doesn't take into consideration the accuracies of the individual sub-taggers. For example, improving XPOS accuracy globally without changing its relative consistency with UPOS will decrease the MEA, indicating that MEA isn't the optimal measure for consistency. Another way of measuring consistency uses *pointwise mutual information* (PMI), or how much more or less often the system gets all three tags correct than one would expect if the three types of accuracy are perfectly uncorrelated. In a baseline system with no mechanism to encourage consistency, UPOS errors and XPOS errors would be expected to occur more or less independently. Letting $\mathbf{c}$ stand in for $\mathbf{c} = \mathbf{y}_i|\mathbf{f} = \mathbf{x}_i$, indicating that the predicted one-hot class vector $\mathbf{c}$ is equal to the gold $\mathbf{y}_i$, it can be shown that the PMI of the two uncorrelated accuracy distributions should be about zero.

$$P(\mathbf{c}^{(\mathrm{u})}, \mathbf{c}^{(\mathrm{x})}) \approx P(\mathbf{c}^{(\mathrm{u})})P(\mathbf{c}^{(\mathrm{x})}) \tag{6.39}$$

$$\frac{P(\mathbf{c}^{(\mathrm{u})}, \mathbf{c}^{(\mathrm{x})})}{P(\mathbf{c}^{(\mathrm{u})})P(\mathbf{c}^{(\mathrm{x})})} \approx 1 \tag{6.40}$$

$$\mathrm{PMI}(\mathbf{c}^{(\mathrm{u})}, \mathbf{c}^{(\mathrm{x})}) \approx 0 \tag{6.41}$$

On the other hand, in a system that does successfully encourage consistency, the probability of the two modules making the correct prediction simultaneously should be greater than the product of the independent probabilities.

$$P(\mathbf{c}^{(\mathrm{u})}, \mathbf{c}^{(\mathrm{x})}) > P(\mathbf{c}^{(\mathrm{u})})P(\mathbf{c}^{(\mathrm{x})}) \tag{6.42}$$

$$\frac{P(\mathbf{c}^{(\mathrm{u})}, \mathbf{c}^{(\mathrm{x})})}{P(\mathbf{c}^{(\mathrm{u})})P(\mathbf{c}^{(\mathrm{x})})} > 1 \tag{6.43}$$

$$\mathrm{PMI}(\mathbf{c}^{(\mathrm{u})}, \mathbf{c}^{(\mathrm{x})}) > 0 \tag{6.44}$$

Thus consistency can be computed straightforwardly given only the four accuracy metrics, because the AllTags metric is the conjunction of the other three.

$$\text{Consistency} = \ln\left(\frac{\text{AllTags}}{\text{UPOS} \cdot \text{XPOS} \cdot \text{UFeats}}\right) \tag{6.45}$$

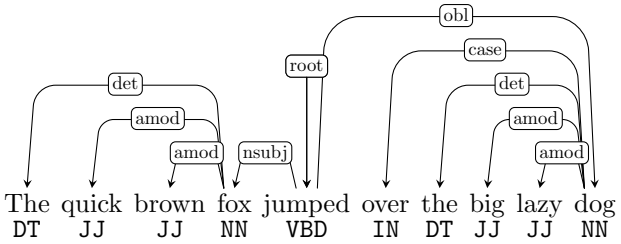
Figure 6.10 shows the effect that the biaffine approach has on consistency compared to the baseline. The mutually exclusive accuracy drops significantly ($p < .001$) when switching to the biaffine tagger; however, this is expected given that the biaffine tagger achieved generally higher performance on XPOS and UFeats tagging. The same outliers in Figure 6.10 are outliers in Figure 6.10a, demonstrating the (undesired) correlation between raw accuracy and mutually exclusive accuracy. The pointwise mutual information, on the other hand, takes into consideration the accuracies of the individual subtasks, immunizing it to the same correlation to raw accuracy. The effect of the biaffine tagger on consistency as measured by PMI in Figure 6.10b is much smaller. The median is significantly greater than zero ($p < .05; V = 484$) according to a signed-rank test, but the mean is only

marginal ($p < .1; t = 1.85$) according to a t-test. This shows that conditioning tagging decisions on each other improves the overall accuracy of the taggers independently, but it also encourages them to make predictions that make sense together.

6.6.2 Distance and linearization

The biaffine approach to parsing laid out in previous discussions only compares intrinsic features of the head and dependent when making predictions. That is, the approach makes no attempt to explicitly model the effects of distance and word order beyond what the RNN is able to learn. One might hypothesize that BiLSTMs are powerful enough to learn and represent linearization and distance information on their own without additional augmentation; but on the other hand, it's possible that explicitly providing the system with these features could improve performance when there are many syntactically and semantically plausible distractors. Consider (6.1).

(6.1)



Even a simple system with only POS information would know that articles depend on nouns, and not on verbs; that is, $P(a_{\tilde{t}} = 1|\text{Dep POS} = \texttt{DET}, \text{Head POS} = \texttt{NOUN}) \gg P(a_{\tilde{t}} = 1|\text{Dep POS} = \texttt{DET}, \text{Head POS} = \texttt{VERB})$. However, in (6.1), there are two determiners and two nouns, meaning that more information is needed to resolve the dependency structure. There are two additional facts about English that the system can leverage here. The first is that in English, determiners precede nouns; and the second is that determiners are normally close to the noun they modify. How can a model capable of explicitly learning these patterns be constructed?

The first step is to formalize the objective. The absolute locations of dependent t and head $\tilde{t}$ probably aren't useful here; instead, the relevant information is the location of t and $\tilde{t}$ *relative* to each other, which is the difference between their indices $\tilde{t} - t$. When the dependent precedes the head, $\text{sign}(\tilde{t} - t) = \ell_{\tilde{t}}$ will be positive; otherwise, it will be negative. Similarly, the absolute value $\text{abs}(\tilde{t} - t) = \delta_{\tilde{t}}$ will indicate the distance between the head and the dependent. There may be relationships for which only the linearization matters and the distance is irrelevant, or vice versa. Thus it may make sense to condition the probability of an edge $a_{\tilde{t}}$ on both of these values independently. This yields the probability objective in Eq. (6.46), which can then be simplified by

assuming conditional independence between the linearization terms $\vec{\ell}$ and distance $\vec{\delta}$ (Eq. 6.47).

$$P(a_{\tilde{t}}|\mathbf{f},\tilde{F},\vec{\ell},\vec{\delta}) = \frac{P(\vec{\ell}|\vec{\delta},\mathbf{f},\tilde{F},a_{\tilde{t}})P(\vec{\delta}|\mathbf{f},\tilde{F},a_{\tilde{t}})\,P(\mathbf{f},\tilde{F}|a_{\tilde{t}})P(a_{\tilde{t}})}{P(\vec{\ell},\vec{\delta},\mathbf{f},\tilde{F})} \tag{6.46}$$

$$= \frac{P(\vec{\ell}|\mathbf{f},\tilde{F},a_{\tilde{t}})\,P(\vec{\delta}|\mathbf{f},\tilde{F},a_{\tilde{t}})P(\mathbf{f},\tilde{F}|a_{\tilde{t}})P(a_{\tilde{t}})}{P(\vec{\ell},\vec{\delta},\mathbf{f},\tilde{F})} \tag{6.47}$$

The expression in Eq. (6.47) is very dense and may need some elucidation. The first two terms in the numerator—$P(\vec{\ell}|a_{\tilde{t}},\mathbf{f},\tilde{F})$ and $P(\vec{\delta}|a_{\tilde{t}},\mathbf{f},\tilde{F})$—are new, having been added in to capture linearization and distance features. The third term in the numerator—$P(\mathbf{f},\tilde{F}|a_{\tilde{t}})$—is the likelihood of the data in a variable-class biaffine classifier, and can be expressed with a deep or shallow biaffine function. The last term of the numerator—$P(a_{\tilde{t}})$—is uniform, and can be ignored. The denominator—$P(\mathbf{f},\tilde{F},\vec{\ell},\vec{\delta})$—ultimately gets used to normalize the softmax function. Putting these together, and making some reasonable conditional independence assumptions, the equation simplifies to Eq. (6.52).

$$s^{(f)}_{t\tilde{t}} = \ln(P(\mathbf{f},\tilde{\mathbf{f}}|a_{\tilde{t}})) \tag{6.48}$$

$$\mathbf{s}^{(f)}_{t} = \text{DeepVCBiaff}(\mathbf{h}_t,\tilde{H}_i) \tag{6.49}$$

$$s^{(\ell)}_{t\tilde{t}} = \ln(P(\ell_{\tilde{t}}|\mathbf{f},\tilde{\mathbf{f}},a_{\tilde{t}})) \tag{6.50}$$

$$s^{(\delta)}_{t\tilde{t}} = \ln(P(\delta_{\tilde{t}}|\mathbf{f},\tilde{\mathbf{f}},a_{\tilde{t}})) \tag{6.51}$$

$$P(a_{\tilde{t}}|\mathbf{f},\tilde{F},\vec{\ell},\vec{\delta}) = \text{softmax}_{\tilde{t}}\left(\mathbf{s}^{(f)}_{t} + \mathbf{s}^{(\ell)}_{t} + \mathbf{s}^{(\delta)}_{t}\right) \tag{6.52}$$

These two new terms have somewhat counterintuitive interpretations. The first is the probability of the linearization given a correct arc, *not* the probability of a correct arc given the linearization. Consider example (6.1); there, a count-based approximation of the probability of an arc between a determiner and a noun given that the determiner precedes the noun is only 67%, because the first instance of *the* precedes both nouns but only depends on one of them. However, in both *gold arcs*, the determiner precedes the noun; thus the probability of $\ell_{\tilde{t}} = 1$ (dependent to the left) given $a_{\tilde{t}}$ is 100%. Similarly, the second term is the probability of the distance given a correct arc. In the example, the probability of an arc between a determiner and a noun given that the determiner is two words away from the noun is 67%, because the second instance of *the* is two words away from both nouns; but *in both correct edges*, the noun is two words away from its determiner. Thus the probability of the distance given an edge provides a stronger signal than the direct probability of an edge given the distance. So the model will try to predict the order and distance between a word t and its possible head $\tilde{t}$ given only their LSTM features and the assumption that there is an edge between them, and penalize the score $s_{t\tilde{t}}$ relative to how incorrect its prediction is. Put another

way, the model asks itself, "if there's an edge between t and $\tilde{t}$, what order should they be in and how close should they be?" and then looks at the actual order and distance. If the model's answer to its self-imposed question differs significantly from the actual order and distance, then it's less likely that there's an edge between them. Yet another, higher-level way to think of this is that the system is trying to learn probabilistic predicate logic rules that govern where edges can appear, like the following:

$$(\texttt{NOUN}(\tilde{t}) \wedge \texttt{DET}(t) \Rightarrow (\texttt{edge}(\tilde{t}, t) \Rightarrow (\tilde{t} > t))) = 1 \tag{6.53}$$

$$(\texttt{NOUN}(\tilde{t}) \wedge \texttt{DET}(t) \Rightarrow (\texttt{edge}(\tilde{t}, t) \Rightarrow (\tilde{t} \not\gg t \wedge \tilde{t} \not\ll t))) = 1 \tag{6.54}$$

These can be read, "if $\tilde{t}$ is a noun and t is a determiner, then if there's an edge between $\tilde{t}$ and t, $\tilde{t}$ must follow t," and "if $\tilde{t}$ is a noun and t is a determiner, then if there's an edge between $\tilde{t}$ and t, $\tilde{t}$ must not be significantly far from t." The system takes in the raw features of the two words $\tilde{t}$ and t, like their parts of speech, and from these produces a probabilistic rule that dictates where the two words should be in relation to each other. Then, if the righthand side of the generated rule is not true—for example, if $\tilde{t} < t$—then it follows that an edge between them is unlikely. This amounts to predicting which order or which distance the words should be in if there's an edge between them, as with the other views of the probability. Critically, this means that the order and distance prediction modules will only be trained on gold arcs, in order to ensure that the system learns e.g. $P(\ell_{\tilde{t}}|\mathbf{f}, \tilde{\mathbf{f}}, a_{\tilde{t}})$—the order of two words with an edge between them—not $P(\ell_{\tilde{t}}|\mathbf{f}, \tilde{\mathbf{f}})$—the order of two arbitrary words in the sentence. The next step is to decide on parametric probability distributions for the two variables.

Linearization

The linearization term can take one of two values, conditioned on both the input features $\mathbf{f}$ and the contextual features $\tilde{\mathbf{f}}$. This means that using a Bernoulli distribution with its single parameter generated as a function of $\mathbf{f}$ and $\tilde{\mathbf{f}}$ is the natural way to approach this probability. The parameter $p^{(\ell)}_{t\tilde{t}}$ must be conditioned on two feature vectors, $\mathbf{f}$ and $\tilde{\mathbf{f}}$; if interactions are permitted between the two feature vectors, then it can be computed as the sigmoid of a deep or shallow fixed-class biaffine function with a single output. The log of this probability—the *linearization score*—is then added

to the base score term, as according to Eqs. (6.50, 6.52).

$$\text{Bernoulli}(x, p) = \exp\Big(x \ln(p) + (1 - x) \ln(1 - p)\Big) \tag{6.55}$$

$$y_{t\tilde{t}}^{(\ell)} = \frac{\text{sign}(\tilde{t} - t) + 1}{2} \tag{6.56}$$

$$p_{t\tilde{t}}^{(\ell)} = \text{sigmoid}(\text{Biaff}(\mathbf{h}_t, \mathbf{h}_{\tilde{t}})) \tag{6.57}$$

$$P(\ell_{\tilde{t}} = y_{t\tilde{t}}^{(\ell)} | \mathbf{f}, \tilde{\mathbf{f}}, a_{\tilde{t}}) = \text{Bernoulli}(y_{t\tilde{t}}^{(\ell)}, p_{t\tilde{t}}^{(\ell)}) \tag{6.58}$$

$$s_{t\tilde{t}}^{(\ell)} = \ln(P(\ell_{\tilde{t}} = y_{t\tilde{t}}^{(\ell)} | a_{\tilde{t}}, \mathbf{f}, \tilde{\mathbf{f}})) \tag{6.59}$$

$$= y_{t\tilde{t}}^{(\ell)} \ln(p_{t\tilde{t}}^{(\ell)}) + (1 - y_{t\tilde{t}}) \ln(1 - p_{t\tilde{t}}^{(\ell)}) \tag{6.60}$$

Note that the model's predicted linearization can be described as $\text{round}(p_{t\tilde{t}}^{(\ell)})$, where—as with $y_{t\tilde{t}}$—1 is "head to the right" and 0 is "head to the left". In predicate logic terms, this is the same as having the system generate the probabilistic rule in Eq. (6.61).

$$((\mathbf{h}_t \wedge \mathbf{h}_{\tilde{t}}) \Rightarrow (\texttt{edge}(t, \tilde{t}) \Rightarrow t < \tilde{t})) = p_{t\tilde{t}}^{(\ell)} \tag{6.61}$$

While the linearization score is used in the edge classifier, it needs to be optimized separately, in order to ensure that the model conditions the probability on gold edges only. Letting $\tilde{y}_t$ be the gold head for dependent t, the linearization loss of word t is simply the cross entropy—the negative score—for its head, shown in Eq. (6.62).

$$\varepsilon_t^{(\ell)} = y_{t,\tilde{y}_t}^{(\ell)} \ln(p_{t,\tilde{y}_t}^{(\ell)}) + (1 - y_{t,\tilde{y}_t}) \ln(1 - p_{t,\tilde{y}_t}^{(\ell)}) \tag{6.62}$$

$$= -s_{t,\tilde{y}_t}^{(\ell)} \tag{6.63}$$

This explicitly shows the model in what contexts heads precede dependents and in what contexts heads follow them, allowing it to generate the correct penalties in the edge scorer. Optimizing the linearization module by backpropagating through the score would be comparable to training the label classifier with the dependent's *predicted* head, rather than its gold one, which allows for inconsistent head/label predictions. The linearization loss is added to the edge loss and the label loss. In sum, parsing decisions can be explicitly conditioned on word order by training an additional module to predict the order of true edges and adding its cross-entropy on possible edges to the raw biaffine score of a potential edge. If the model judges two words to be in the wrong order for an edge to exist between them—such as a noun that precedes a determiner—then it is discouraged from assigning a dependency arc.

Distance

Incorporating distance is a bit trickier. So far, the discussions have been restricted to categorical features and classes, but the distance term now under consideration is numeric. Consequently, neither a Bernoulli nor a Categorical distribution will work here, and it turns out that there are a variety of potential options. One possible probability model for the distance between two words $P(\delta_{\tilde{t}}|\mathbf{f}, \tilde{F}, a_{\tilde{t}})$ is a Poisson distribution, which is single-parameter, discrete, and bounded to be in the interval $[0, \infty)$. Additionally, larger distances will have higher variance, which reflects the intuition that it's more permissible to misjudge the distance by five or six words when there's a larger gap between head and dependent. The basic approach would involve generating a Poisson distribution over all possible distances between t and $\tilde{t}$, and then seeing how likely the true distance actually is according to the model. The Poisson distribution is given in Eq. (6.64), with its rate parameter λ being the mean of the distribution, and—when λ is an integer—the mode.

$$\text{Poisson}(x, \lambda) = \frac{\lambda^x \exp(-\lambda)}{x!} \tag{6.64}$$

While the Poisson distribution has a number of advantages, it has some disadvantages as well; most of these can be worked around or tolerated, with one exception. The biggest issue that arises with the Poisson model is that the penalty for making an error scales exponentially with the size of the error $\lambda_{t\tilde{t}}^{(\delta)} - y_{t\tilde{t}}^{(\delta)}$. That is, the score scales more or less linearly with the error (though it's slightly more than linear for edges that are too long), but the score is ultimately exponentiated by the softmax function. This can result in severe instability during training.

$$\ln(\text{Poisson}(x, \lambda)) = x \ln(\lambda) - \lambda - \ln(x!) \tag{6.65}$$

Ultimately, all that's needed is a function that converts two vectors and an integer into a penalty, so one could attempt to modify the Poisson probability mass function in order to address this concern. This might work empirically, but there are too many possible functions one could propose to consider them all, and the ideal situation would be to use an approach that makes clear and interpretable assumptions. So one might think the Poisson distribution might be a good fit for modeling the likelihood of arc lengths; but while it does have some appealing properties, in practice it penalizes unexpectedly long distances much too harshly.

Another initially appealing approach to modeling the distance stably involves utilizing a different distribution entirely. This other approach is to have the model place a Gaussian distribution on the possible distances. Ultimately the Gaussian distribution will be exchanged for a very similar but less common distribution, but the intuition here will hold for that one as well. The mean μ then would be the distance that the system is the most confident is correct, and the variance would be fixed at some constant σ^2, or left as a learnable parameter independent of the input. Technically, the normal distribution has a problem that any continuous distribution will have—the probability of the

variable taking any individual value $\delta_{\tilde{t}}$ is infinitessimally small. However, the softmax normalization renders this a non-issue because its output must sum to one, allowing it to distinguish between infinitessimally small values.[6]

$$\text{Gaussian}(x, \mu, \sigma^2) = \frac{1}{\sqrt{\tau\sigma^2}} \exp\left(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right) \tag{6.66}$$

$$P(\delta_{t\tilde{t}}|a_{\tilde{t}}, \mathbf{f}, \tilde{F}) = \text{Gaussian}(y_{t\tilde{t}}^{(\delta)}, \mu_{t\tilde{t}}^{(\delta)}, \sigma^2) \tag{6.67}$$

$$\ln\left(P(\delta_{t\tilde{t}}|a_{\tilde{t}}, \mathbf{f}, \tilde{F})\right) = \ln\left(\text{Gaussian}(y_{t\tilde{t}}^{(\delta)}, \mu_{t\tilde{t}}^{(\delta)}, \sigma^2)\right) \tag{6.68}$$

$$s_{t\tilde{t}}^{(\delta)} = -\frac{1}{2}\left(\ln(\tau\sigma^2) + \frac{(\mu_{t\tilde{t}}^{(\delta)} - y_{t\tilde{t}}^{(\delta)})^2}{\sigma^2}\right) \tag{6.69}$$

Unfortunately, this maintains the exponential penalty; the score is quadratic in $(\mu_{t\tilde{t}}^{(\delta)} - y_{t\tilde{t}}^{(\delta)})$, which then gets exponentiated in the softmax function. Exponential penalties would appear to be a common theme of the exponential family of distributions, suggesting that a distribution outside that family may be ideal in this situation. The Cauchy distribution is one such non-exponential distribution, having a very similar shape to the Gaussian distribution. In fact, it can even be parameterized to have a probability density function expressed very similarly to the Gaussian distribution.

$$\text{Gaussian}(x, \mu, \sigma^2) = \frac{1}{\sqrt{\tau\,\sigma^2}} \exp\left(\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right)^{-1} \tag{6.70}$$

$$\text{Cauchy}(x, \mu, \sigma^2) = \frac{1}{\sqrt{\frac{\tau^2}{2}\sigma^2}} \left(1+\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right)^{-1} \tag{6.71}$$

Critically, the Cauchy distribution has a quadratic rather than exponential penalty, resulting in a long tail. For large differences in $\mu_{t\tilde{t}}^{(\delta)}$ and $y_{t\tilde{t}}^{(\delta)}$ where the 1 is negligible, the only meaningful difference between the Gaussian-based score in Eq. (6.69) and the Cauchy-based score in Eq. (6.74) is the logarithm in the Cauchy score. This logarithm cancels out with the exponentiation in the softmax, which is what yields the quadratic rather than exponential penalty. This makes it a natural way to address the exponentially decreasing probability assigned by the Gaussian distribution to unusually long or short edges.

$$P(\delta_{t\tilde{t}}|a_{\tilde{t}}, \mathbf{f}, \tilde{F}) = \text{Cauchy}(y_{t\tilde{t}}^{(\delta)}, \mu_{t\tilde{t}}^{(\delta)}, \sigma^2) \tag{6.72}$$

$$\ln\left(P(\delta_{t\tilde{t}}|a_{\tilde{t}}, \mathbf{f}, \tilde{F})\right) = \ln\left(\text{Cauchy}(y_{t\tilde{t}}^{(\delta)}, \mu_{t\tilde{t}}^{(\delta)}, \sigma^2)\right) \tag{6.73}$$

$$s_{t\tilde{t}}^{(\delta)} = -\left(\frac{1}{2}\ln\left(\frac{\tau^2}{2}\sigma^2\right) + \ln\left(1 + \frac{1}{2}\frac{(\mu_{t\tilde{t}}^{(\delta)} - y_{t\tilde{t}}^{(\delta)})^2}{\sigma^2}\right)\right) \tag{6.74}$$

[6] $\tau = 2\pi$

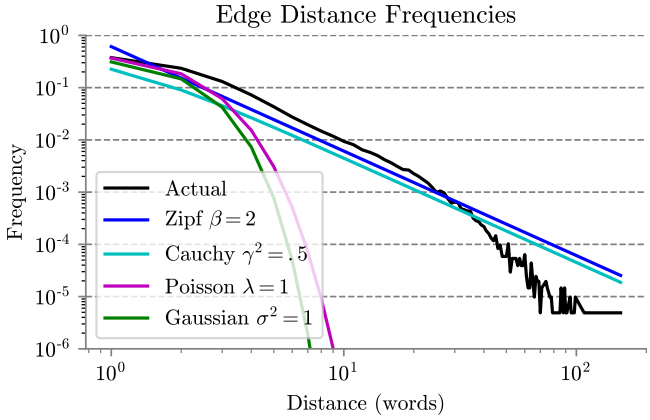


Figure 6.11: Distribution of arc distances in the English Web Treebank. Shown on a log-log scale. The non-exponential Zipf and Cuachy distributions fit the data much better than the exponential distributions.

The Cauchy distribution does not technically have a defined mean or variance; but the parameterization in Eq. (6.71) uses parameters analogous to the mean and variance parameters of the Gaussian distribution, so in this case the Gaussian distribution can be swapped for the Cauchy distribution with no other changes.

Not only is this a very numerically stable approach to conditioning edge predictions on relative distances, but it also turns out to more accurately reflect the prior distribution of edges in the Universal Dependencies corpus. This is shown for the English Web Trebank in Figure 6.11. The Poisson, Gaussian, and Cauchy distributions are graphed with their canonical probability density functions and $\mu = 0$. Edge distances can be seen to follow a roughly Zipf-Mendelbrot distribution, the probability mass function of which is shown in Eq. (6.75).[7]

$$\text{Zipf-Mendelbrot}(x, \alpha, \beta) = \frac{1}{\zeta}\left(\frac{x^\beta}{\alpha}\right)^{-1} \tag{6.75}$$

The Zipf-Mendelbrot distribution is closely related to the Cauchy distribution (consider the case where $\beta = 2$ and $\alpha = 2\sigma^2$), but its mode cannot be shifted, making it unsuitable for predicting distances. The Cauchy distribution can be conceptualized as a version of the Zipf distribution that has been modified to allow the mode to shift. Because of this relationship, a Cauchy distribution is an observably better fit to the prior probability of edge distances than a Poisson or Gaussian distribution, which assigns unreasonably low probability to unexpectedly long edges. This further suggests that the Cauchy distribution would be well suited to modeling the edge likelihood given

[7] In Eq. (6.75), ζ is a normalization constant

the features.

As with the linearization term, this module must be trained independently of the edge classifier. The obvious way to train it is to maximize the probability of the correct distance of each gold head. Normally one would use an L_2 loss to learn a real value; but the L_2 loss imposes the assumption that the conditional probability of the output variable follows a Gaussian distribution. Since it's been argued that the conditional probability of the edge distances follows a Cauchy distribution, it's more sensible here to use the *log* L_2 loss, which is simply the negative distance score of t's gold head.

$$\varepsilon_t^{(\delta)} = -\ln\left(1 + \frac{(y_{t,\tilde{y}_t} - \mu_{t,\tilde{y}_t})^2}{2\sigma^2}\right) \tag{6.76}$$

$$= -s_{t,\tilde{y}_t}^{(\delta)} \tag{6.77}$$

Complete edge classifier

The algorithm for the whole edge classifier, including the linearization and distance modules, is summarized below.

$$y_{t\tilde{t}}^{(\ell)} = \frac{\text{sign}(\tilde{t} - t) + 1}{2} \tag{6.78}$$

$$y_{t\tilde{t}}^{(\delta)} = |\tilde{t} - t| \tag{6.79}$$

$$s_{t\tilde{t}}^{(f)} = \text{DeepBiaff}(\mathbf{h}_t, \mathbf{h}_{\tilde{t}}) \tag{6.80}$$

$$p_{t\tilde{t}}^{(\ell)} = \text{sigmoid}(\text{DeepBiaff}(\mathbf{h}_t, \mathbf{h}_{\tilde{t}})) \tag{6.81}$$

$$s_{t\tilde{t}}^{(\ell)} = y_{t\tilde{t}}^{(\ell)} \ln(p_{t\tilde{t}}^{(\ell)}) + (1 - y_{t\tilde{t}}^{(\ell)}) \ln(1 - p_{t\tilde{t}}^{(\ell)}) \tag{6.82}$$

$$\mu_{t\tilde{t}}^{(\delta)} = \text{softplus}(\text{DeepBiaff}(\mathbf{h}_t, \mathbf{h}_{\tilde{t}})) \tag{6.83}$$

$$s_{t\tilde{t}}^{(\delta)} = -\ln\left(1 + \frac{1}{2}\frac{(\mu_{t\tilde{t}}^{(\delta)} - y_{t\tilde{t}}^{(\delta)})^2}{\sigma^2}\right) \tag{6.84}$$

$$\mathbf{s}_t^{(\text{e})} = \mathbf{s}_t^{(f)} + \text{StopGrad}(\mathbf{s}_t^{(\ell)}) + \text{StopGrad}(\mathbf{s}_t^{(\delta)}) \tag{6.85}$$

$$\hat{\mathbf{y}}_t^{(\text{e})} = \text{softmax}(\mathbf{s}_t^{(\text{e})}) \tag{6.86}$$

$$\varepsilon_t^{(\text{e})} = -\mathbf{y}_t^{\top(\text{e})}(\ln(\hat{\mathbf{y}}_t^{(\text{e})}) + \mathbf{s}_t^{(\ell)} + \mathbf{s}_t^{(\delta)}) \tag{6.87}$$

Eqs. (6.78, 6.79) define the target values of the linearization and distance modules. Eq. (6.80) defines the base score of the edge from word $\tilde{t}$ to t. Eqs. (6.81, 6.82) defines the linearization score, which is the model's attempt to answer the question "are these words in the right order for there to be an edge from $\tilde{t}$ to t?" Eqs. (6.83, 6.84) defines the distance score, where the model asks "are these words close enough together or far enough apart for there to be an edge from $\tilde{t}$ to t?" Eqs. (6.85, 6.86) combines the raw score, the linearization score, and the distance score into one edge score and then applies the softmax function to turn it into a probability. Note that the linearization and distance scores

should not be backpropagated through here, indicated by the StopGrad function—error should only be backpropagated to the raw score. Eq. (6.87) defines the error term to be minimized, which is the sum of the log probability of the gold edge under a Categorical distribution, the log probability of the observed linearization of the gold edge under a Bernoulli distribution, and the log (normalized) probability of the observed distance of the gold edge under a Cauchy distribution.

6.6.3 Results

These discussions explained in depth how to incorporate the relative locations of two words into the edge classifier. The new classifier includes a module that learns to predict the order of words in a gold edge, and another that learns to predict the distance between those words. The entropy of these modules is then used to augment the raw biaffine score. Having motivated this approach mathematically, what impact do these additional features make on final performance? Figure 6.12 shows the macro-averaged difference in LAS on the CoNLL 2018 datasets between the baseline and parsers with explicit relative location information. Languages without validation data are excluded. Adding the linearization module can be seen to make a small but statistically significant difference in mean ($t = -2.5; p < .05$) and median ($V = 640; p < .05$) performance. One might expect that adding the linearization score would make a large improvement, given the importance of linearization to most linguistic theory. However, adding the linearization term makes only a small improvement over the baseline. This could indicate that the basic system is already very good at capturing word order information, which seems reasonable given the sequential nature of the LSTM. It could also reflect the unordered nature of the exact linguistic representation that the UD annotation scheme is designed to model, since the functional structure of Lexical Functional Grammar doesn't have the same notion of headedness that the constituency structures have. Alternatively, it could be that the linearization score didn't assign a high enough penalty to make an impact. This final possibility could be addressed in future work by introducing positive weights onto the linearization and distance scores and either tuning the weights by hand or by leaving them as trainable parameters in the model.

The distance module makes a more substantial mean and median impact ($p < .001$), and having both modules together yields the largest performance gain ($p < .001$). The improvement from the distance prediction suggests that the baseline system may be getting confused on longer sentences when there are more semantically feasible head words; building in a way to explicitly identify which of the feasible head words are closer and which are farther may allow it to narrow down the possible heads more efficiently. This can be tested by plotting the accuracy of the different variations for different edge lengths and different sentence lengths. Figure 6.13 provides these plots. Systems with a distance module actually appear to slightly underperform ones without on longer edges; this can be attributed to them picking up on the abundance of short arcs in the dependency tree and overgeneralizing. On the other hand, distance-augmented systems perform noticeably better on

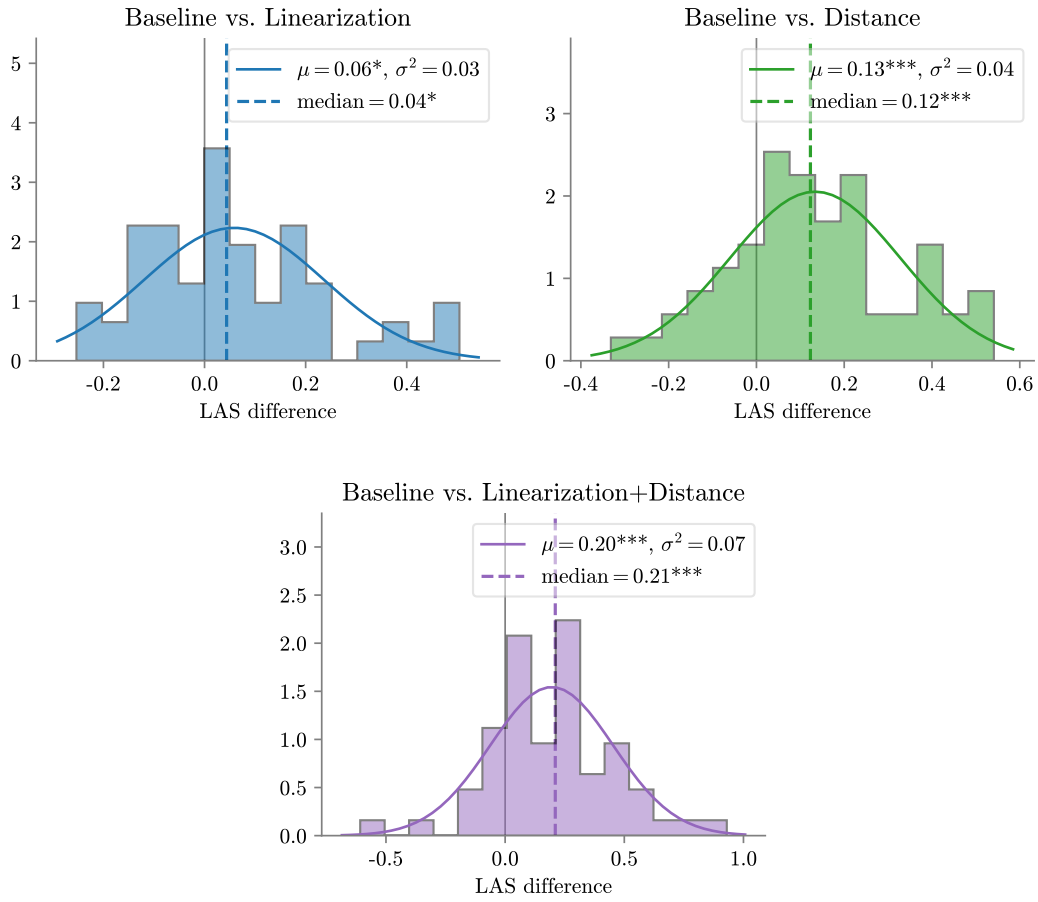


Figure 6.12: Distance/linearization ablation, evaluated on total accuracy. Difference in LAS for systems with the basic biaffine scorer, systems with a linearization module, systems with a distance module, and systems with both. Statistically significant differences from the baseline are marked with asterisks. The linearization module makes a slight improvement, but the distance module is fairly substantial.

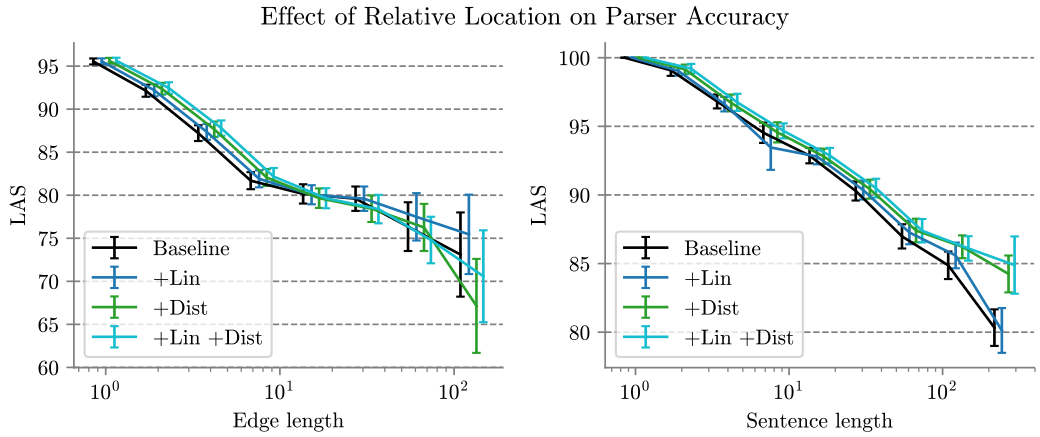


Figure 6.13: Distance/linearization ablation, evaluated by sentence length. Labeled accuracy of the four system variants across different gold edge lengths (left) and sentence lenghts (right). The distance module improves accuracy on longer sentences.

extremely long sentences, suggesting that the additional distance module helps to avoid distractors. The fact that primarily extremely long sentences benefit may reflect that the distance penalty is actually not quite harsh enough. As with the linearization score, adding a learned parameter that the system can use to increase the distance score may help the system to take further advantage of this additional information.

This section has proposed a mathematically and empricially well-motivated approach to incorporating distance and linearization into variable-class classifiers over sequences. The approach was applied to dependency parsing, but in principle it can be applied to any task that can be cast as involving variable-class classifiers, including tasks like neural machine translation that take advantage of attention mechanisms.

6.6.4 Other CoNLL 2018 extensions

The system that achieved the highest performance on the CoNLL 2018 shared task, Che et al. (2018), built heavily on the parser described in this thesis. They used two powerful techniques to improve accuracy: firstly, they outfitted the system with ELMo (Peters et al., 2018), a recently proposed alternative to pretrained word embeddings; secondly, they used model ensembling to make predictions from multiple trained systems. ELMo involves training an LSTM language model on an extremely large corpus. In downstream applications, the language model is run on a sentence to process, with the hidden features of the LSTM then being extracted for each word and used as embeddings. Che et al. (2018) augmented the tagger and parser with ELMo embeddings, finding that it improved tagging performance by 0.56% and parsing performance by 0.84%. The parser ensemble averaged the probabilities predicted by three systems, using the resulting distribution to

make final parsing decisions. This likewise improved performance over a baseline, especially for smaller treebanks, averaging a 0.55% improvement.

(Straka, 2018), who achieved the second-highest performance, modify the described model by sharing subsets of the weights in the tagger and parser. They examine two variants: a *loosely joint* one only shares the word embeddings (defined as the concatenation of a pretrained embedding, a trainable whole-token embedding, and a character-based LSTM embedding), and a *tightly joint* one that additionally shares the BiLSTM weights. They found that the loosely joint model performed slightly better on tagging (improving UPOS, XPOS, and UFeats by .057% on average, and AllTags by .09%), whereas the tightly joint model performed slightly better on parsing (outperforming in UAS and LAS by .2%). Since the multitask setup can be conceptualized as a way of regularizing the system, this suggests that the dependency parser is more prone to overfitting than the tagger (either by virtue of the task or the particular hyperparameter configuration).

(Kanerva et al., 2018) modified the tagger to include UFeats, but found that the "embarrassingly" simple approach of concatenating the UFeats feature set into the XPOS tags worked better than more complex approaches. That is, rather than predicting an XPOS tag and a UFeats feature set, they designed the system to predict a single atomic XPOS+UFeats tag. They found that conditioning parsing decisions on the XPOS+UFeats tags instead of the XPOS tags improved parsing performance for a subset of languages with simplex XPOS tags but many morphological features, whereas this approach had little effect on languages with XPOS tags that contained more or less the same information as the UFeats. They point to concatenating the UFeats features with the UPOS tag instead as a direction of future research, since this will minimize redundancy and avoids the slight inconsistency of concatenating universal features to a language-specific part of speech tag.

6.7 Conclusion

This chapter recapitulated the relatively simple neural system for parsing described in Chapter 5, and extended it to achieve state-of-the-art performance on the 2017 CoNLL Shared Task on UD parsing without utilizing ensembling or language model pretraining. It provided some further analysis of the original system, comparing the relative performance of nonprojective arc-factored and transition-based architectures on this task. It found evidence that modern arc-factored parsers might be better at producing nonprojective arcs (though this claim is not without some caveats). Additionally, this system performs better when there's an abundance of data, suggesting that more regularization could improve accuracy on lower-resource languages. This chapter also added several new components to the system and then examined how they impact performance. For the 2017 shared task, it added an analogous POS tagger and a character-level word embedding model. For the 2018 shared task, it added a method for conditioning universal features and language-specific POS tags on universal POS tags and a method for conditioning edges on the relative locations of

two words.

This chapter also sought to quantitatively justify the additional complexity of the new components over the simpler variant in Chapter 5. It examined how important the POS tagger is to the system, comparing the downstream performance of parsers using the proposed tagger, the baseline tagger, and no tagger at all. The proposed tagger beats both baselines significantly, whereas the two baselines in fact don't statistically differ from each other, indicating that POS tags can help this system but must be sufficiently accurate. The character-based approach was found to significantly boost performance on languages that scored high on the metric for morphological complexity—both for parsing and tagging—suggesting that constructing token representation from subtoken information is effective for capturing the influence of morphology on syntax, and the naïve approach of using only holistic word embeddings is insufficient. The success at the shared task demonstrates that a well-tuned, straightforward neural approach to parsing and tagging can get state-of-the-art performance for datasets with a wide variety of syntactic properties.

This chapter made further improvements to the system for the CoNLL 2018 shared task. Modifying the POS tagger to condition features and language-specific tags on the system's own universal tag predictions was found to not only slightly improve the consistency of the different tag metrics, but also to globally improve the accuracy of the other tags. Including linearization and especially distance features in the system made further improvements over the baseline, with the distance feature proving particularly useful in extremely long sentences.

The system proposed in Chapter 5 and extended for the CoNLL 2017 and 2018 shared tasks has already made an impact in the world of dependency parsing. Many other teams used the system as a starting point for their submission to the CoNLL 2018 shared task, including the three highest-performing teams, who made relatively small changes (if any) to the original system.

The following chapter extends the parser once more, this time aiming at a slightly different parsing paradigm where a word can have more than one head word, or potentially none at all.

Chapter 7

Extension to Semantic Dependencies

7.1 Introduction

The previous two chapters described in detail a neural approach to parsing basic Stanford Dependencies (De Marneffe et al., 2006) and Universal Dependencies (Nivre et al., 2016). Both of these formalisms are tree-structured, which makes them easy for transition-based parsers to parse. However, the tree structure restriction sacrifices some potentially critical information about the sentence. For example, in the sentence *Sandy wants to buy a book*, the word *Sandy* is the subject of both *want* and *buy*—either or both relationships could be useful in a downstream task, but a strictly tree-structured representation of this sentence (as in Figure 7.1a) can only represent one of them. A number of formalisms inspired by syntactic fameworks other than Lexical Functional Grammar (Kaplan and Bresnan 1982; see Chapter 2) have been developed to have richer representations that can capture more information in a single sentence.

Because of this, the collapsed SD and enhanced UD representations (SD+/UD+) relax the tree-structure constraint. Parsing the syntactic frameworks directly normally requires brittle, hand-engineered grammars and formalisms-specific parsers, as explained in Chapter 4; and parsing the simplified graph-structured dependency representations is very difficult for transition-based approaches, which are optimized for generated a much more restrictive class of structures. This chapter explores how to adapt the arc-factored system described above so that it can produce other graph-structured formalisms with as few changes as possible. Achieving high performance on graph-structured formalisms as well as tree-structured formalisms would further legitimize the techniques proposed in this thesis.

The 2014 SemEval shared task on Broad-Coverage Semantic Dependency Parsing (Oepen et al.,

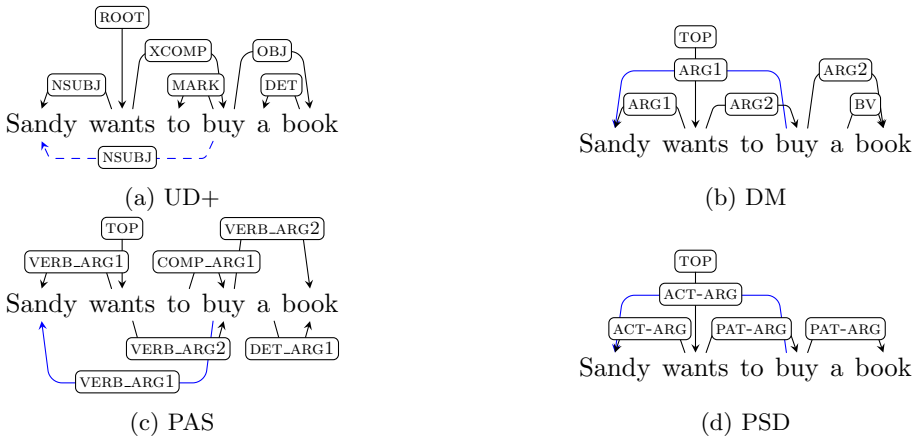


Figure 7.1: Comparison between syntactic and semantic dependency schemes.

2014) introduced three dependency representations that do away with the assumption of strict tree structure in favor of a richer graph-structured representation, allowing them to capture more linguistic information in a sentence. The graph structure opens up the possibility of providing more useful information to downstream tasks (Reddy et al., 2017; Schuster et al., 2017), but increases the difficulty of automatically extracting that information. It is more difficult not only because most recent work on dependency parsing has focused on generating tree-structured formalisms, but also because the formalism is less restrictive—that is, in a dependency tree, knowing that word i depends on word j entails that it doesn't depend on word $k \neq i, j$, but in a dependency graph, knowing that i depends on j provides no information on whether it depends on k as well. The SemEval datasets are ideal for testing the graph-structured formalism because there are a number of strong baselines for them, and they introduce no parsing complications beyond their graph-structured nature (as opposed to UD+, which also introduces empty nodes).

Chapter 5 described a successful syntactic dependency parsing system with few task-specific sources of complexity. This chapter, relating work published in Dozat and Manning (2018), extends that system to be able to train on and produce the graph-structured data of semantic dependency schemes. It also considers straightforward extensions of the system that are likely to increase performance over the straightforward baseline. Chapter 6 found that part-of-speech tags made a significant positive impact on performance; since (gold) lemmas are provided with the dataset, and they add linguistic information complementary to part-of-speech tags, it's worth investigating what impact they make on performance as well. Chapter 6 also found that building in a character-level word embedding model was helpful for many languages and wouldn't depend on an external lemmatizer that could propagate errors. Lastly, this chapter briefly examines some of the design choices of that architecture, in order to assess which components are necessary for achieving the highest accuracy and which have little impact on final performance.

7.2 Background

7.2.1 Semantic dependencies

The 2014 SemEval (Oepen et al., 2014, 2015) shared task introduced three new semantic dependency formalisms, applied to the Penn Treebank (shown in Figure 7.1), compared to enhanced Universal Dependencies (UD+; Nivre et al. 2016): DELPH-IN MRS (DM; Flickinger et al. 2012; Oepen and Lønning 2006); Predicate-Argument Structures (PAS; Miyao and Tsujii 2004); and Prague Semantic Dependencies (PSD; Hajic et al. 2012). Whereas syntactic dependencies generally annotate grammatical relationships between words—such as *subject* and *object*—semantic dependencies aim to reflect semantic relationships—such as *agent* and *patient* (cf. semantic role labeling (Gildea and Jurafsky, 2002)). The SemEval semantic dependency schemes are directed acyclic graphs (DAGs) instead of trees, allowing them to annotate function words as being heads (following modern semantic theory) without lengthening paths between content words (as in 7.1c). This is slightly more restrictive than UD+, which allows cycles in some constructions.

7.2.2 Related work

This approach to semantic dependency parsing builds off the work from Chapters 5 and 6 at syntactic dependency parsing and Peng et al. (2017) at semantic dependency parsing. In these approaches, parsing involves first using a multilayer bidirectional LSTM over word and part-of-speech tag embeddings. Parsing is then done using directly-optimized self-attention over recurrent states to attend to each word's head (or heads), and labeling is done with an analogous multi-class classifier.

Peng et al.'s (2017) system uses a max-margin classifer on top of a BiLSTM, with the score for each graph coming from several sources. First, it scores each word as either taking dependents or not. Then, for each ordered pair of words, it scores the arc from the first word to the second. Lastly, it scores each possible labeled arc between the two words. The graph that maximizes these scores may not be consistent—with an edge coming from a non-predicate, for example—so they enforce hard constraints in order to prune away invalid semantic graphs. Decisions are not independent, so in order to find the highest-scoring graph that follows these constraints, they use the AD^3 decoding algorithm (Martins et al., 2011).

The approach in Chapter 5 to syntactic dependency parsing is similar to their approach, but avoids the possibility of generating internally inconsistent trees by fully factorizing the system. Rather than summing the scores from multiple modules and then finding the valid structure that maximizes that sum, the system makes parsing and labeling decisions sequentially, choosing the labels for each edge only after the edges in the tree have been finalized by an MST algorithm.

Wang et al. (2018) take a different approach in their recent work, using a transition-based parser built on stack-LSTMs (Dyer et al., 2015). They extend Choi and McCallum's (2013) transition system for producing non-projective trees so that it can produce arbitrary DAGs and they modify

the stack-LSTM architecture slightly to make the network more powerful.

7.3 Approach

7.3.1 Basic approach

The semantic dependency parsing task can be formulated as labeling each edge in a directed graph, with *null* being the label given to pairs with no edge between them. Using only one module that labels each edge in this way would be an *unfactorized* approach. However, it can also be factorized into two modules: one that predicts whether or not a directed edge (w_j, w_i) exists between two words, and another that predicts the best label for each potential edge.

The approach described here closely follows the approach to tree-structured dependency parsing of Dozat and Manning (2017), described in Chapter 4. As with many successful recent parsers, word and POS tag embeddings are concatenated, and fed into a multilayer bidirectional LSTM to get richer representations. Formally, letting sentence i be composed of T embeddings $\mathbf{x}_{it}$, the recurrent layer is as in Eq. (7.1).

$$H_i = \text{BiLSTM}(X_i) \tag{7.1}$$

To avoid notational clutter, the sentence and timestep indices it will be dropped in subsequent expressions. For each of the two modules—the *edge* module (e) that predicts which words have edes, and *label* module (l) that generates the best label for every edge—two single-layer feedforward networks (FFNN) split the top recurrent states into two parts—a *dependent* representation (notated without tilde), as in Eq. (7.2, 7.3), and a *head* representation (notated with tilde), as in Eq. (7.4, 7.5). This allows for reduced recurrent sizes, which helps to avoid overfitting in the classifer without weakening the LSTM's representational capacity.

$$\mathbf{v}_t^{(\text{e})} = \text{FFNN}(\mathbf{h}_t) \qquad \text{Edge-dep} \tag{7.2}$$

$$\mathbf{v}_t^{(\text{l})} = \text{FFNN}(\mathbf{h}_t) \qquad \text{Label-dep} \tag{7.3}$$

$$\tilde{\mathbf{v}}_t^{(\text{e})} = \text{FFNN}(\mathbf{h}_t) \qquad \text{Edge-head} \tag{7.4}$$

$$\tilde{\mathbf{v}}_t^{(\text{l})} = \text{FFNN}(\mathbf{h}_t) \qquad \text{Label-head} \tag{7.5}$$

Bilinear or biaffine classifiers—which are generalizations of linear classifiers to include multiplicative interactions between two vectors, motivated in Chapter 3—predict edges and labels. The bilinear/ biaffine classifiers can be *single-class* (SC; Eqs. 7.6, 7.7), generating a single score to be used in a binary classifier, or *multi-class* (MC; 7.8, 7.9), generating multiple scores to be used in a categorical

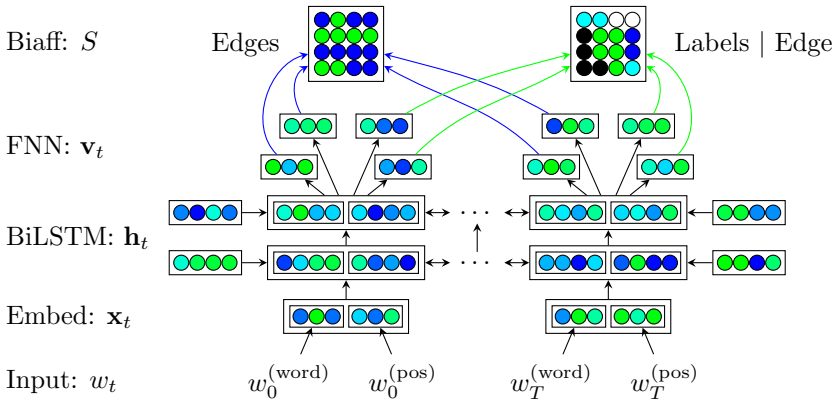


Figure 7.2: The basic architecture of the factorized system. Word embeddings $\mathbf{x}_t$ are fed into a BiLSTM, which uses FNNs to get four separate representations $\mathbf{v}_t$ for each word. Two biaffine scorers turn these into score matrices S, where each cell represents the score of an edge or label between two tokens.

classifier.

$$\text{SCBilin}(\mathbf{x}, \tilde{\mathbf{x}}) = \mathbf{x}^\top U \tilde{\mathbf{x}} \tag{7.6}$$

$$\text{SCBiaff}(\mathbf{x}, \tilde{\mathbf{x}}) = \mathbf{x}^\top U \tilde{\mathbf{x}} + \mathbf{w}^\top (\mathbf{x} \oplus \tilde{\mathbf{x}}) + b \tag{7.7}$$

$$\text{MCBilin}(\mathbf{x}, \tilde{\mathbf{x}}) = \mathbf{x}^\top \mathbf{U} \tilde{\mathbf{x}} \tag{7.8}$$

$$\text{MCBiaff}(\mathbf{x}, \tilde{\mathbf{x}}) = \mathbf{x}^\top \mathbf{U} \tilde{\mathbf{x}} + W(\mathbf{x} \oplus \tilde{\mathbf{x}}) + \mathbf{b} \tag{7.9}$$

$$s_{t,\tilde{t}}^{(\text{e})} = \text{SCBiaff}\left(\mathbf{v}_t^{(\text{e})}, \tilde{\mathbf{v}}_{\tilde{t}}^{(\text{e})}\right) \tag{7.10}$$

$$\mathbf{s}_{t,\tilde{t}}^{(\text{l})} = \text{MCBiaff}\left(\mathbf{v}_t^{(\text{l})}, \tilde{\mathbf{v}}_{\tilde{t}}^{(\text{l})}\right) \tag{7.11}$$

The tensor U in the single-class variant or $\mathbf{U}$ in the multi-class one can optionally be diagonal (such that $u_{jkj'} = 0$ wherever $j \neq j'$) to conserve parameters. The unlabeled parser (trained with sigmoid cross-entropy) scores every edge between pairs of words in the sentence—these scores can be decoded into a graph by keeping only edges that received a positive score. The labeler (trained with softmax cross-entropy) scores every label for each pair of words, so it simply assigns each edge predicted by the edge classifier its highest-scoring label. The system is trained according to the sum of these two losses, with error backpropagating to the labeler only through gold edges. This system is shown graphically in Figure 7.2. Unfortunately, sometimes the loss for one module overwhelms the loss for the other, causing the system to underfit. Including a tunable interpolation constant $\lambda \in (0, 1)$ successfully evens out the two error terms $\varepsilon^{(\text{e})}$ and $\varepsilon^{(\text{l})}$.

$$\varepsilon = (1 - \lambda)\varepsilon^{(\text{e})} + \lambda\varepsilon^{(\text{l})} \tag{7.12}$$

Worth noting is that the removal of the maximum spanning tree algorithm and change from softmax cross-entropy loss to weighted sigmoid cross-entropy in the unlabeled parser represent the only changes needed to allow the original syntactic parser in Chapter 4 to generate fully graph-structured semantic dependency output. Note also that this system is general enough that it could be used for any graph-structured dependency scheme, including the enhanced dependencies of the Universal Dependencies formalism (with the exception of gapped constructions, which are absent from the SemEval datasets).

7.3.2 Comparison with Peng et al

In the original syntactic parser, the problem of finding the best labeled incoming edge for each dependent was factorized into two steps: finding the best head for each dependent, and then labeling the dependent based on the information provided by the predicted head. Formally, let $g_{\tilde{t}k}$ (for *graph*) be a categorical variable representing the edge from token $\tilde{t}$ to a given, fixed token t with label k. The matrix G of all the $g_{\tilde{t}k}$ variables will be one-hot, since each token has exactly one head and one label in the syntactic setup. The tree-structured dependency parser splits $g_{\tilde{t}k}$ into two separate variables: $a_{\tilde{t}}$, representing the *row* of the correct class in G, and c_k, representing the *column* of the correct class. The single probability of the two variables can then be equivalently represented as the product of two separate probabilities of one variable each.

$$P(g_{\tilde{t}k} = 1|\mathbf{f}, \tilde{F}) = P(c_k = 1, a_{\tilde{t}} = 1|\mathbf{f}, \tilde{F}) \tag{7.13}$$

$$= \frac{P(c_k, a_{\tilde{t}}, \mathbf{f}, \tilde{F})}{P(\mathbf{f}, \tilde{F})} \tag{7.14}$$

$$= \frac{P(c_k, a_{\tilde{t}}, \mathbf{f}, \tilde{F})}{P(\mathbf{f}, \tilde{F})} \frac{P(a_{\tilde{t}}, \mathbf{f}, \tilde{F})}{P(a_{\tilde{t}}, \mathbf{f}, \tilde{F})} \tag{7.15}$$

$$= \frac{P(c_k, a_{\tilde{t}}, \mathbf{f}, \tilde{F})}{P(a_{\tilde{t}}, \mathbf{f}, \tilde{F})} \frac{P(a_{\tilde{t}}, \mathbf{f}, \tilde{F})}{P(\mathbf{f}, \tilde{F})} \tag{7.16}$$

$$= P(c_k|a_{\tilde{t}}, \mathbf{f}, \tilde{F}) P(a_{\tilde{t}}|\mathbf{f}, \tilde{F}) \tag{7.17}$$

$$= P(c_k|a_{\tilde{t}}, \mathbf{f}, \tilde{\mathbf{f}}_{\tilde{t}}) P(a_{\tilde{t}}|\mathbf{f}, \tilde{F}) \tag{7.18}$$

Eq. (7.13) splits the matrix variable into a row variable and a column variable. Eq. (7.14) expands the definition of conditional probability. Eq. (7.15) multiplies by $\frac{x}{x}$. Eq. (7.16) swaps the denominators of the fractions. Eq. (7.17) reapplies the definition of conditional probability, proving that the two expressions are the same. Eq. (7.18) assumes conditional independence between c_k and $\tilde{\mathbf{f}}_{\tilde{t}' \neq \tilde{t}}$ give $a_{\tilde{t}}$—that is, features of the alternative candidate heads don't influence the probability of the label of the dependent. An advantage of this approach is that it makes imposing hard constraints that only refer to one variable easier. If the tree where each word depends on its highest scoring head is

malformed, it can be corrected into a single-root maximum spanning tree without having to take into consideration how the choice of label affects the probability of the head. An additional advantage is that the two probabilities can be computed with classifiers that use different hyperparameters. While this increases the hyperparameter space, it also allows for more finely-tuned systems. Finally, the independence assumptions of the factorized approach has implications for optimization. Under a standard softmax cross-entropy loss objective, the unfactorized approach during training will aim to reduce the probability of *all* non-gold edge/label pairs $g_{\tilde{t}k}$ according to how much probability mass the model assigned them. The factorized approach only decreases the probability of incorrect labels for the correct edge, without touching the probabilities of incorrect labels for incorrect edges. One might expect that the factorized approach will be easier to optimize because it won't put extra resources into reducing the probabilities of labeled edges pruned away by the edge classifier, but this hypothesis warrants additional experimentation.

The approach to graph-structured dependency parsing laid out in this chapter straightforwardly extends this idea. In a graph-structured dependency representation, the matrix variable G is no longer one-hot, because a token can have multiple heads. Now, each row of G is either null or one-hot, meaning G can't be split into a "row" variable and a "column" variable as before. Instead, a vector of variables $\mathbf{a}$ that are strictly less informative than G will be introduced. The new variables will indicate whether or not a row of G has a 1 in it, meaning that there is an edge from word $\tilde{t}$ to t but providing no information about the label of that edge. This additional variable allows the probability to be factorized in the same way as the tree-structure parser, though this time no additional independence assumptions are actually being made. For consistency of notation, G will also be renamed C where $\mathbf{a}$ is introduced.

$$P(g_{\tilde{t}k} = 1|\mathbf{f}, \tilde{F}) = P(c_{\tilde{t}k} = 1, a_{\tilde{t}} = 1|\mathbf{f}, \tilde{F}) \tag{7.19}$$

$$= P(c_{\tilde{t}k}|a_{\tilde{t}}, \mathbf{f}, \tilde{F})P(a_{\tilde{t}}|\mathbf{f}, \tilde{F}) \tag{7.20}$$

This mimics some of the advantages in the original parser: post-processing constraints on structure are easier to enforce; the two modules can use different hyperparameters; and optimization may be slightly easier and more efficient.

(Peng et al., 2017) likewise used an arc-factored BiLSTM parser to tackle the SemEval datasets. However, their approach lacks some of the theoretical and practical advantages of the biaffine approach presented here. Their system predicts three things from the BiLSTM state: given one token $\tilde{t}$, whether it's a predicate capable of taking dependents $p_{\tilde{t}}$ (p); given two tokens t and $\tilde{t}$, whether there's an edge (arc) between them $a_{\tilde{t}}$ (e); and given two tokens, what labeled edge (if any) is between them $c_{\tilde{t}k}$ (le). However, their system doesn't condition each successively more complex prediction on the output of previous predictions, as the systems proposed in this thesis do. Instead, they simply multiply the independent probabilities together. Formally, they insert two redundant

variables into the original objective (Eq. 7.21) and rewrite it into the product of three probabilities (Eq. 7.22)—similar to this work—but then they assume conditional independence between the variables (Eq. 7.23).

$$P(g_{\tilde{t}k}|\mathbf{f},\tilde{F}) = P(\,c_{\tilde{t}k}, a_{\tilde{t}}, p_{\tilde{t}}\,|\mathbf{f},\tilde{F}) \tag{7.21}$$

$$= P(\,c_{\tilde{t}k}|a_{\tilde{t}}, p_{\tilde{t}}\,, \mathbf{f},\tilde{F})P(\,a_{\tilde{t}}|p_{\tilde{t}}\,, \mathbf{f},\tilde{F})P(\,p_{\tilde{t}}\,|\tilde{\mathbf{f}},\tilde{F}) \tag{7.22}$$

$$\approx P(\,c_{\tilde{t}k}\,|\mathbf{f},\tilde{F})P(\,a_{\tilde{t}}\,|\mathbf{f},\tilde{F})P(\,p_{\tilde{t}}\,|\tilde{\mathbf{f}},\tilde{F}) \tag{7.23}$$

The additional independence assumption (which clearly doesn't hold, since the variables are partially redundant) means that the new expression no longer maintains exact equivalence to the original objective. Additionally, by removing the redundant conditioning variables, the system can no longer simplify postprocessing steps to ensure well-formed structures; in fact, ensuring that the different modules don't produce contradictory predictions makes finding the best dependency graph at inference time *more* rather than *less* complicated.

Another difference from Peng et al.'s (2017) approach is that they use feedforward variable-class classifiers rather than biaffine ones to compute the probabilities for $c_{\tilde{t}k}$ and $a_{\tilde{t}}$. That is, in their approach, the final score for an edge from token $\tilde{t}$ to token t with label k is computed using three feedforward scorers. For clarity, the weight vectors $\mathbf{u}$ are all distinct, and none of them depend on anything in the input sequence i.

$$s^{(\mathrm{p})}_{\tilde{t}} = \mathbf{u}^\top \mathrm{FFNN}(\mathbf{h}_{\tilde{t}}) \tag{7.24}$$

$$s^{(\mathrm{e})}_{\tilde{t}t} = \mathbf{u}^\top \mathrm{FFNN}(\mathbf{h}_t \oplus \mathbf{h}_{\tilde{t}}) \tag{7.25}$$

$$s^{(\mathrm{le})}_{t\tilde{t}k} = \mathbf{u}_k^\top \mathrm{FFNN}(\mathbf{h}_t \oplus \mathbf{h}_{\tilde{t}}) \tag{7.26}$$

$$s_{t\tilde{t}k} = s^{(\mathrm{le})}_{t\tilde{t}k} + s^{(\mathrm{e})}_{t\tilde{t}} + s^{(\mathrm{p})}_{\tilde{t}} \tag{7.27}$$

Chapter 5 argued against the FFNN approach to variable-class classification on theoretical and empirical grounds. In this instance, their system is missing some logically possible (and theoretically necessary) terms. The feedforward term $\mathrm{FFNN}(\mathbf{h}_t \oplus \tilde{\mathbf{t}})$ in Peng et al.'s model and the bilinear term $\mathbf{h}_{\tilde{t}}^\top \mathbf{U}\mathbf{h}_t$ in ours are both ways of modeling nonlinear interactions between the head and dependent. Abstracting away from the exact scoring function that gets applied to the BiLSTM hidden states and emphasizing which representations are interacting (the dependent state $\mathbf{h}_t$, the head state $\mathbf{h}_{\tilde{t}}$, and label weights k), Peng et al.'s scorer is shown in Eq. (7.28). The factorized system in this chapter uses two separate scorers, which are not added together—one to score each possible edge

Eq. (7.29), and one to compute each possible label for each predicted edge Eq. (7.30).

$$s_{t\tilde{t}k} = f^{(k)}(\mathbf{h}_t, \mathbf{h}_{\tilde{t}}) + f(\mathbf{h}_t, \mathbf{h}_{\tilde{t}}) + f(\mathbf{h}_{\tilde{t}}) \quad \text{PTS17} \quad (7.28)$$

$$s_{t\tilde{t}} = f(\mathbf{h}_t, \mathbf{h}_{\tilde{t}}) + f(\mathbf{h}_t) + f(\mathbf{h}_{\tilde{t}}) + f() \quad \text{DM18 (Edges)} \quad (7.29)$$

$$s_{t\tilde{t}k} = f^{(k)}(\mathbf{h}_t, \mathbf{h}_{\tilde{t}}) + f^{(k)}(\mathbf{h}_t) + f^{(k)}(\mathbf{h}_{\tilde{t}}) + f^{(k)}() \quad \text{DM18 (Labels)} \quad (7.30)$$

$f^{(k)}$ means that the function is parameterized for label k. Functions with no arguments represent constant bias terms. In Eq. (7.29), the score for each unlabeled edge comes from the full range of possible constant, first-order, and second-order terms, and likewise in Eq. (7.30). However, Peng et al.'s scorer in Eq. (7.28) uses only the two interaction terms $f^{(k)}(\mathbf{h}_t, \mathbf{h}_{\tilde{t}})$ and $f(\mathbf{h}_t, \mathbf{h}_{\tilde{t}})$ and one non-interaction term $f(\mathbf{h}_{\tilde{t}})$. Not only does this mean that their scorer is slightly less powerful than the one proposed here, but it's also not perfectly sound theoretically. As shown in Chapter 3, the lower-order terms are necessary for ensuring that the only assumption needed to make Eq. (7.31) true is conditional independence between input features.

$$P(c_{\tilde{t}k}|\mathbf{f}, \tilde{F}) = \text{softmax}_k(\mathbf{s}_{it\tilde{t}}) \quad (7.31)$$

To briefly recapitulate Chapter 3, when features are binary and modeled with categorical distributions, the lower-order terms absorb the probability of the class when the higher-order feature interactions are absent (i.e. $P(c_k|(f_j \wedge f_{j'}) = 0)$). When features are continuous and thus modeled with continuous distributions, the story can be a bit more complex, but the lower-order terms still perform similar roles. If the full range of terms are included in the score $s_{\tilde{t}k}$, then the probability of the labeled edge given the features simplifies to the softmax of the score vector. Conversely, if the full range of terms are not included in the score, then the probability of the labeled edge does *not* actually simplify to the simple softmax of the score, meaning that the softmax classifier is technically inappropriate for the task. On the one hand, neural networks are normally powerful enough to be able to compensate for this theoretical inefficiency, but on the other hand, it's not clear why one should favor a theoretically less sound approach over a theoretically sound one that achieves similar or better performance.

This discussion is to show that the basic approach to semantic dependency parsing used by Peng et al. lacks many of the appealing properties that the biaffine parser has. It should be noted though that they include a number of techniques not included in this work. Specifically, they explore two orthogonal approaches to multitask parsing across the three different semantic dependencies datasets. In the basic multitask version, they train a system with a shared BiLSTM that parses all three formalisms simultaneously. In one extension to this, they use *frustratingly easy domain adaptation* (Daumé III, 2007; Kim et al., 2016), where the top layer of the shared BiLSTM is concatenated with the top layer of a task-specific BiLSTM as well. In another, they use

a tensor-product scorer (a higher-order generalization of the biaffine one motivated here) designed to ensure consistent predictions across the three tasks. Their scorer uses a decomposed sixth-order tensor to model this, but in principle it could be done with a third-order one. For each module (predicate, edges, labeled edges), the outer product of the vector representation $\mathbf{v}_{t\tilde{t}} = \text{FFNN}(\mathbf{h}_t, \mathbf{h}_{\tilde{t}})$ for each of the three datasets is weighted and summed in order to get the score. This allows the system to learn relationships among the three representations. The simple shared BiLSTM approach to multitasking makes no apparent improvement over the single-task system, but adding in either the task-specific BiLSTM states or the third-order decoder improves the micro average by about a quarter of a percent, with the improvement seemingly additive. Either or both multitask additions could in principle help the biaffine system, although neither of them has been tried at this point.

7.3.3 Augmentations

Ballesteros et al. (2016), Dozat et al. (2017) (described in Chapter 6), and Ma et al. (2018) find that character-level word embedding models improve performance for syntactic dependency parsing, so it seems sensible to explore the impact it has on semantic dependency parsing. Dozat et al. (2017) confirm that the syntactic parser performs better with POS tags, which raises the question of whether word lemmas—another form of low-level lexical information—might also improve dependency parsing performance. Section 7.4.2 below explores both of these options for semantic dependency parsing.

7.4 Results

7.4.1 Hyperparameters

	Hidden Sizes	**Drop Prob**
Embedding	100	20%
GloVe linear	125	0%
Char LSTM	1 @ 400	33%/33% (FF/recur)
Char linear	100	33%
BiLSTM	3 @ 600	45%/25% (FF/recur)
Arc/Label FFNN	600/600	25%/33%

Loss & Optimizer	
Interpolation (λ)	.025
L_2 regularization	$3e^{-9}$
Learning rate	$1e^{-3}$
Adam β_1	0
Adam β_2	.95

Table 7.1: Final hyperparameter configuration of the semantic dependency parser.

	DM		PAS		PSD		Avg	
	ID	OOD	ID	OOD	ID	OOD	ID	OOD
Du et al. (2015)	89.1	81.8	91.3	87.2	75.7	73.3	85.3	80.8
Almeida and Martins (2015)	88.2	81.8	90.9	86.9	76.4	74.8	85.2	81.2
WCGL18	90.3	84.9	91.7	87.6	78.6	75.9	86.9	82.8
PTS17: Basic	89.4	84.5	92.2	88.3	77.6	75.3	87.4	83.6
PTS17: Freda3	90.4	85.3	92.7	89.0	78.5	76.4	88.0	84.4
Ours: Basic	92.4	87.6	**94.0**	90.8	80.1	78.0	89.6	86.3
Ours: +Char	92.6	88.0	**94.0**	90.7	80.3	78.2	89.7	86.4
Ours: +Lemma	**93.6**	**88.9**	93.9	90.7	80.9	79.1	**90.2**	86.9
Ours: +Char +Lemma	93.4	88.8	**94.0**	**91.0**	**81.0**	**79.2**	**90.2**	**87.0**

Table 7.2: Semantic dependency parsing performance.
Comparison between the proposed system and the previous state of the art on in-domain (WSJ) and out-of-domain (Brown corpus) data, according to labeled F1 (LF1).

The hyperparameters for the basic system (that is, without character embeddings or lemmas) were tuned fairly extensively on the DM development data. The hyperparameter configuration for the final system is given in Table 7.1. All input embeddings (word, pretrained, POS, etc.) are concatenated, and the system uses the gold POS tags and lemmas provided with the datasets. The pretrained GloVe embeddings are 100-dimensional (Pennington et al., 2014), but linearly transformed to be 125-dimensional to allow the system to rearrange the embedding space in a way that's optimized for the high degree of dropout. Only words or lemmas that occurred 7 times or more are included in the word and lemma embedding matrix; including less frequent words appears to facilitate overfitting. Character-level word embeddings are generated using a one-layer unidirectional LSTM that convolved over three character embeddings at a time, whose end state is linearly transformed to be 100-dimensional. The core BiLSTM is three layers deep. The different types of word embeddings—word, GloVe, and character-level—are dropped simultaneously during training, but independently from POS and lemma embeddings (which are likewise dropped independently of each other). This is done to encourage the different kind of word embeddings from learning representations that are too similar, defeating the purpose of dropout. Dropped embeddings are replaced with learned `<DROP>` tokens. LSTMs use same-mask recurrent dropout (Gal and Ghahramani, 2016), applied in both "horizontal" connections and "vertical" ones. The systems are trained with batch sizes of 3,000 tokens for up to 75,000 training steps, terminating early after 10,000 steps pass with no improvement in validation accuracy.

7.4.2 Performance

Table 7.2 compares the system's performance with the alternative systems. The system uses biaffine classifiers (Eqs. 7.7, 7.9), with no ReLU nonlinearities in the FFNN layers, and a diagonal $\mathbf{U}$ tensor in the label classifier (Eq. 7.10) but a full, square U tensor in the edge classifier (Eq. 7.11). The

system trains at a speed of about 300 sequences/second on an NVIDIA Titan X and parses about 1,000 sequences/second. Du et al. (2015) and Almeida and Martins (2015) are the systems that won the 2015 shared task (closed track). *PTS17: Basic* represents the single-task versions of Peng et al. (2017), which they make multitask across the three datasets in Freda3 by adding frustratingly easy domain adaptation and the third-order decoding mechanism. WCGL18 is Wang et al.'s (2018) transition-based system. The fully factorized basic system already substantially outperforms both Peng et al.'s single-task baseline and their much more complex multi-task approach. Including a character-level word embedding model (similar to Dozat et al.'s (2017)) can improve performance slightly, and providing gold lemma embeddings likewise improves performance even further. These effects appear additive, so that including both together generally pushes performance even higher. Many infrequent words were excluded from the frequent token embedding matrix, so it makes sense that the system should improve when provided more lexical information that's harder to overfit on.

Surprisingly, the PAS dataset seems not to benefit substantially from lemma or character embeddings. It has been noted that PAS is the easiest of the three datasets to achieve good performance for; so one possible explanation is that 94% LF1 may simply be near the ceiling of what can be achieved for the dataset without using unsupervised pretraining methods on large unlabeled corpora. Alternatively, the main difference bewteen PAS and DM/PSD is that PAS includes semantically vacuous function words in its representation. Because function words are extremely frequent, it's possible that they are being disproportionately represented in the loss or LF1 score. Using a hinge loss (like Peng et al. (2017)) instead of a cross-entropy loss might help, since the system would stop focusing on potentially "easy" functional predicates once it learned to predict their argument structures confidently, allowing it to pour more resources into modeling more challenging phenomena. Similarly, the loss interpolation constant λ was tuned on DM, and may be suboptimal for the other datasets.

7.4.3 Variations

This section examines the impact that slight variations on the architecture have on final performance in Figure 7.3. Twenty models were trained on the DM treebank for each variation under consideration, keeping all other hyperparameters constant. Rank-sum tests (Lehmann et al., 1975) reveal that the basic system outperforms variants with no hidden layers in the edge classifier ($W = 389; p < .001$) or the label classifier ($W = 396.5; p < .001$). Diagonal $\mathbf{U}$ tensors ($W = 384; p < .001$), the ReLU nonlinearity ($W = 386, p < .001$), and to a lesser extent omitting the linear terms ($W = 266, p < .1$) seem to weaken the edge classifier.

The unfactorized system performed as well as the factorized one, indicating that the system could be simplified even further. This is not terribly surprising though, since they are mathematically equivalent (see Eqs. 7.13–7.17) and here used the same hyperparameters. The improved performance of deeper systems (replicating the findings in Chapter 5) likely justifies their added complexity. On the other hand, the choice between biaffine and bilinear classifiers largely comes down to aesthetics,

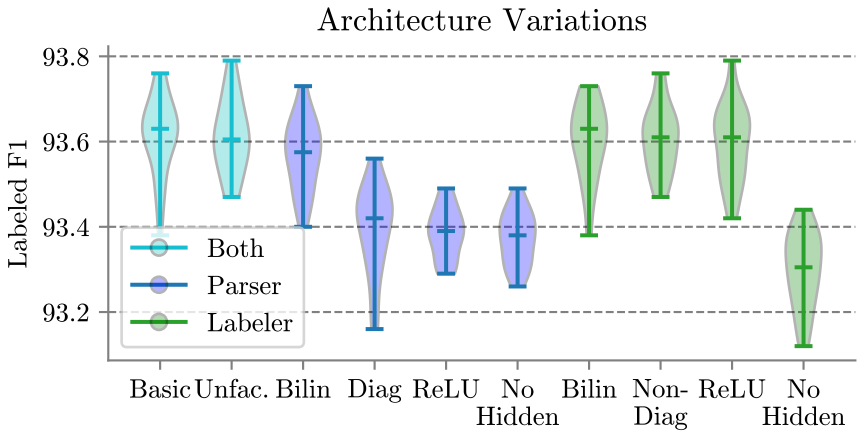


Figure 7.3: Performance of architecture variations for the semantic dependency parser. Basic+lemma system; unfactorized (labeler-only); with bilinear classifiers (Eqs. 7.6, 7.8); with nondiagonal $\mathbf{U}$ in the labeler or diagonal U in the parser; with the ReLU nonlinearity; ommitting the hidden layers (Eqs. 7.4–7.3).

since the change from biaffine to bilinear represents only a small decrease in overall power. The fact that ReLU makes no difference to the label classifier but badly hurts the edge classifier is difficult to explain; perhaps ReLU is stripping away too much information critical for unlabeled parsing when it sparsifies the representations, or perhaps removing the negative numbers in the head and dependent representation prevents more complex interactions that are useful to the bilinear component. Forcing the bilinear matrix U in the edge classifier to be diagonal also seems to simplify the biaffine function too much, taking away power needed for the complex edge discrimination task.

Overall, the labeler displayed considerable invariance to architecture changes, while the edge classifier proved to be more fragile; however, the biggest drop in median performance represented only .3% LF1. Since this system is larger and more regularized than the other systems, this relative robustness to architecture variations suggests that unglamorous, low-level hyperparameters—such as hidden sizes and dropout rates—can be as critical to performance as high-level architecture enhancements.

7.5 Discussion

This chapter minimally extended the simple syntactic dependency parser from Chapter 5 to produce graph-structured dependencies. With only a few tweaks and careful tuning, this system achieves state-of-the-art performance, further highlighting the generality of the biaffine architecture motivated at great length in this thesis. The high-performing arc-factored parser can be adapted to different types of dependency graphs (projective tree, non-projective tree, directed graph) with only

small changes without obviously hurting accuracy. By contrast, transition-based parsers—which were originally designed for parsing projective constituency trees, and have very different structures (Nivre, 2003; Aho and Ullman, 1972)—require whole new transition sets or even data structures to generate arbitrary graphs. Arguably, this points to arc-factored parsers like the one in this thesis being the most natural way to produce dependency graphs with different structural restrictions.

Furthermore, this work demonstrates that a multitask system relying on a complex decoding algorithm to prune away invalid graph structures isn't necessary for achieving the level of parsing performance a simpler system can achieve (though in principle it could push performance even higher). It also finds easier or independently motivated ways to improve accuracy—taking advantage of provided lemma information provides a boost comparable to one found by drastically increasing system complexity (though getting this improvement in real-world situations depends on having an accurate lemmatizer), and paying close attention to hyperparameters ensures that the system performs as best it can. Thus the biaffine arc-factored parser can efficiently achieve excellent performance on a wide variety of datasets without requiring significant additional complexity.

Chapter 8

Conclusion

This thesis has laid out a straightforward, fast, accurate, and theoretically sound neural dependency parser. It proved that the primary architecture innovations follow mathematically from the same principles that yield the standard affine softmax classifier, unlike the more common feedforward classifiers. The LSTM-based parser was carefully tuned and shown to achieve high accuracy, nearing the state-of-the-art performance achieved by a much more complex system. The system is also relatively fast to run at inference time, making it feasible for use as a black-box in non-academic settings. Comparing the proposed arc-factored system against a transition-based baseline reveals evidence suggesting that the arc-factored approach is more effective at generalizing crossing dependencies in languages with more free word order than the transition-based approach. In addition to using an arc-factored parser, it was found that using a character-level word embedding model improved performance on languages where grammatical functions are dictated more by orthographically-inferrable morphology than strict word order. Similarly, incorporating components designed to explicitly model the effects of distance and word order on the probability of there being a head-dependent relationship between two words was shown to help accuracy on longer sentences with more distractors. The actual parsing and label classification processes closely mirror more basic sequence labeling, meaning that only small changes are needed to change the parser into a part-of-speech tagger; then the part-of-speech tagger itself was made to take advantage of multiplicative interactions between neural features using similar bilinear components to the ones used in the main parser, pointing to the potential ubiquity of the innovations made in this paper. In addition to adapting the tree-structure parser to function as a tagger, this work showed how to adapt it to function as a graph-structure parser with only a few minimal changes to the architecture. Ultimately, the parser at the core of this work finds an excellent balance between being fast, accurate, theoretically satisfying, flexible, and easy-to-extend, having already influenced several other lines of research on parsing and other related tasks.

Bibliography

Steven P Abney and Mark Johnson. 1991. Memory requirements and local ambiguities of parsing strategies. *Journal of Psycholinguistic Research* 20(3):233–250.

Alfred V Aho and Jeffrey D Ullman. 1972. *The theory of parsing, translation, and compiling*, volume 1. Prentice Hall.

Chris Alberti, David Weiss, and Slav Petrov. 2015. Improved transition-based parsing and tagging with neural networks. In *EMNLP*.

Mariana SC Almeida and André FT Martins. 2015. Lisbon: Evaluating TurboSemanticParser on multiple languages and out-of-domain data. In *SemEval*. pages 970–973.

Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. 2016. Globally normalized transition-based neural networks. In *ACL*. volume 1, pages 2442–2452.

Giuseppe Attardi. 2006. Experiments with a multilanguage non-projective dependency parser. In *CoNLL*. pages 166–170.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. In *ICLR*.

James K Baker. 1979. Trainable grammars for speech recognition. *The Journal of the Acoustical Society of America* 65(S1):S132–S132.

Miguel Ballesteros, Chris Dyer, and Noah A Smith. 2015. Improved transition-based parsing by modeling characters instead of words with LSTMs. In *EMNLP*.

Miguel Ballesteros, Yoav Goldberg, Chris Dyer, and Noah A Smith. 2016. Training with exploration improves a greedy stack-LSTM parser. In *EMNLP*.

Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2016. Enriching word vectors with subword information. In *EMNLP*.

Taylor L Booth and Richard A Thompson. 1973. Applying probability measures to abstract languages. *IEEE transactions on Computers* 100(5):442–450.

Johan Bos. 1996. Predicate logic unplugged. In *10th Amsterdam Colloquium*.

Joan Bresnan. 1995. Morphology competes with syntax: Explaining typological variation in weak crossover effects. *Is the best good enough* pages 59–92.

Joan Bresnan. 2001. *Lexical-functional syntax*, volume 16. Blackwell Oxford.

Kris Cao and Marek Rei. 2016. A joint model for word embedding and word morphology. In *ACL*. pages 18–26.

Xavier Carreras. 2007. Experiments with a higher-order projective dependency parser. In *EMNLP-CoNLL*.

Arun Tejasvi Chaganty, Ashwin Paranjape, Jason Bolton, Matthew Lamm, Jinhao Lei, Abigail See, Kevin Clark, Yuhao Zhang, Peng Qi, and Christopher D Manning. 2017. Stanford at TAC KBP 2017: Building a trilingual relational knowledge graph. In *TAC*.

Wanxiang Che, Yijia Liu, Yuxuan Wang, Bo Zheng, and Ting Liu. 2018. Towards better UD parsing: Deep contextualized word embeddings, ensemble, and treebank concatenation. In *CoNLL 2018 Shared Task*. ACL, pages 55–64. http://www.aclweb.org/anthology/K18-2005.

Danqi Chen and Christopher D Manning. 2014. A fast and accurate dependency parser using neural networks. In *EMNLP*. pages 740–750.

Hongshen Chen, Yue Zhang, and Qun Liu. 2016. Neural network for heterogeneous annotations. In *EMNLP*. pages 731–741.

Hao Cheng, Hao Fang, Xiaodong He, Jianfeng Gao, and Li Deng. 2016. Bi-directional attention with agreement for dependency parsing. *EMNLP* .

Kyunghyun Cho, Bart Merrienboer, Caglar Gulcehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *EMNLP*.

Jinho D Choi and Andrew McCallum. 2013. Transition-based dependency parsing with selectional branching. In *ACL*. volume 1, pages 1052–1062.

Noam Chomsky. 1957. *Syntactic structure*. Mouton.

Noam Chomsky. 1965. *Aspects of the Theory of Syntax*. MIT press.

Noam Chomsky. 1970. Remarks on nominalization. In Roderick Jacobs and Peter Rosenbaum, editors, *Reading in English transformational Grammar*, Blaisdell.

Noam Chomsky. 1986. *Knowledge of language: Its nature, origin, and use*. Greenwood Publishing Group.

Noam Chomsky. 1994. *The minimalist program*. MIT Press, Cambridge, Massachusetts.

Yoeng-Jin Chu and Tseng-Hong Liu. 1965. On shortest arborescence of a directed graph. *Scientia Sinica* 14(10):1396.

Kevin Clark, Minh-Thang Luong, Christopher D. Manning, and Quoc V. Le. 2018. Semi-supervised sequence modeling with cross-view training. In *EMNLP*.

John Cocke and Jacob Schwarts. 1970. Programming languages and their compilers: Preliminary notes. Ms. (CIMS, NYU).

Michael Collins. 2003. Head-driven statistical models for natural language parsing. *Computational linguistics* 29(4):589–637.

Ann Copestake, Dan Flickinger, Carl Pollard, and Ivan A Sag. 2005. Minimal recursion semantics: An introduction. *Research on Language and Computation* 3(2-3):281–332.

Michael A Covington. 2001. A fundamental algorithm for dependency parsing. In *ACM Southeast*. Citeseer, pages 95–102.

Michal Daniluk, Tim Rocktäschel, Johannes Welbl, and Sebastian Riedel. 2017. Frustratingly short attention spans in neural language modeling. In *ICLR*.

Hal Daumé III. 2007. Frustratingly easy domain adaptation. In *ACL*. pages 256–263.

Marie-Catherine De Marneffe, Bill MacCartney, Christopher D Manning, et al. 2006. Generating typed dependency parses from phrase structure parses. In *LREC*. volume 6, pages 449–454.

Marie-Catherine De Marneffe and Christopher D Manning. 2008. The Stanford typed dependencies representation. In *Coling 2008*. pages 1–8.

David Dowty. 1991. Thematic proto-roles and argument selection. *Language* 67(3):547–619.

Timothy Dozat and Christopher D. Manning. 2017. Deep biaffine attention for neural dependency parsing. *ICLR* .

Timothy Dozat and Christopher D. Manning. 2018. Simpler but more accurate semantic dependency parsing. In *ACL*.

Timothy Dozat, Peng Qi, and Christopher D Manning. 2017. Stanford's graph-based neural dependency parser at the CoNLL 2017 shared task. *CoNLL 2017 Shared Task* pages 20–30.

Yantao Du, Fan Zhang, Xun Zhang, Weiwei Sun, and Xiaojun Wan. 2015. Peking: Building semantic dependency graphs with a hybrid parser. In *SemEval*. pages 927–931.

John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research* 12:2121–2159.

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A Smith. 2015. Transition-based dependency parsing with stack long short-term memory. In *EMNLP*.

Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A Smith. 2016. Recurrent neural network grammars. In *NAACL*. pages 199–209.

Jay Earley. 1970. An efficient context-free parsing algorithm. *Communications of the ACM* 13(2):94–102.

Jack Edmonds. 1967. Optimum branchings. *Journal of Research of the national Bureau of Standards B* 71(4):233–240.

Jason Eisner. 1996. Efficient normal-form parsing for combinatory categorial grammar. In *ACL*. pages 79–86.

Joseph E Emonds. 1976. *A transformational approach to English syntax: Root, structure-preserving, and local transformations*. Academic Press New York.

Ronald Aylmer Fisher. 1930. *The genetical theory of natural selection: a complete variorum edition*. Oxford University Press.

Dan Flickinger, Yi Zhang, and Valia Kordoni. 2012. Deepbank. a dynamically annotated treebank of the Wall Street Journal. In *International Workshop on Treebanks and Linguistic Theories*. pages 85–96.

Lyn Frazier. 1987. Sentence processing: A tutorial review. *Attention and performance 12: The psychology of reading* pages 559–586.

Harold N Gabow, Zvi Galil, Thomas Spencer, and Robert E Tarjan. 1986. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica* 6(2):109–122.

Yarin Gal and Zoubin Ghahramani. 2016. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *ICML*.

Daniel Gildea and Daniel Jurafsky. 2002. Automatic labeling of semantic roles. *Computational linguistics* 28(3):245–288.

Jonathan Ginzburg and Ivan A. Sag. 2000. *Interrogative Investigations: The Form, Meaning, and Use of English Interrogatives*. CSLI Publications, Stanford.

Yoav Goldberg and Joakim Nivre. 2012. A dynamic oracle for arc-eager dependency parsing. *COLING* pages 959–976.

Yoav Goldberg and Joakim Nivre. 2013. Training deterministic parsers with non-deterministic oracles. *ACL* 1:403–414.

Alex Graves and Jürgen Schmidhuber. 2005. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks* 18(5-6):602–610.

Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. 2016. LSTM: A search space odyssey. In *IEEE Transactions on Neural Networks and Learning Systems*.

Jarrod D Hadfield. 2010. Mcmc methods for multi-response generalized linear mixed models: The MCMCglmm R package. *Journal of Statistical Software* 33(2):1–22.

Jan Hajič, Massimiliano Ciaramita, Richard Johansson, Daisuke Kawahara, Maria Antònia Martí, Lluís Màrquez, Adam Meyers, Joakim Nivre, Sebastian Padó, Jan Štěpánek, et al. 2009. The CoNLL-2009 shared task: Syntactic and semantic dependencies in multiple languages. In *CoNLL*. pages 1–18.

Jan Hajic, Eva Hajicová, Jarmila Panevová, Petr Sgall, Ondrej Bojar, Silvie Cinková, Eva Fucíková, Marie Mikulová, Petr Pajas, Jan Popelka, et al. 2012. Announcing Prague Czech-English dependency treebank 2.0. In *LREC*. pages 3153–3160.

Kazuma Hashimoto, Yoshimasa Tsuruoka, Richard Socher, et al. 2017. A joint many-task model: Growing a neural network for multiple nlp tasks. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. pages 1923–1933.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. pages 770–778.

Luheng He, Kenton Lee, Mike Lewis, and Luke Zettlemoyer. 2017. Deep semantic role labeling: What works and whats next. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. volume 1, pages 473–483.

Harold Stanley Heaps. 1978. *Information retrieval: Computational and theoretical aspects*. Academic Press, Inc.

Gustav Herdan. 1960. *Type-token mathematics*, volume 4. Mouton.

Geoffrey E Hinton. 2012. A practical guide to training restricted boltzmann machines. In Montavon G., Orr G.B., and Mller KR, editors, *Neural networks: Tricks of the trade*, Springer, pages 599–619.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9(8):1735–1780.

Richard A Hudson. 1984. *Word grammar*. Blackwell Oxford.

Ray Jackendoff. 1977. *X-bar syntax*. The MIT Press.

Peng Jin, Yue Zhang, Xingyuan Chen, and Yunqing Xia. 2016. Bag-of-embeddings for text classification. In *IJCAI*. volume 16, pages 2824–2830.

Jenna Kanerva, Filip Ginter, Niko Miekka, Akseli Leino, and Tapio Salakoski. 2018. Turku neural parser pipeline: An end-to-end system for the CoNLL 2018 shared task. In *CoNLL 2018 Shared Task*. pages 133–142. http://www.aclweb.org/anthology/K18-2013.

Ronald M. Kaplan and Joan Bresnan. 1982. Lexical-functional grammar: A formal system for grammatical representation. In Joan Bresnan, editor, *The Mental Representation of Grammatical Relations*, MIT Press, Cambridge, Massachusetts, MIT Press Series on Cognitive Theory and Mental Representation, chapter 4, pages 173–281.

Tadao Kasami. 1966. An efficient recognition and syntax-analysis algorithm for context-free languages. In *Coordinated Science Laboratory Report no. R-257*. Coordinated Science Laboratory, University of Illinois at Urbana-Champaign.

Young-Bum Kim, Karl Stratos, and Ruhi Sarikaya. 2016. Frustratingly easy neural domain adaptation. In *COLING*. pages 387–396.

Diederik Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *ICLR*.

Eliyahu Kiperwasser and Yoav Goldberg. 2016. Simple and accurate dependency parsing using bidirectional LSTM feature representations. In *ACL*. volume 4, pages 313–327.

Dan Klein and Christopher D Manning. 2003. Accurate unlexicalized parsing. In *ACL*. volume 1, pages 423–430.

Terry Koo and Michael Collins. 2010. Efficient third-order dependency parsers. In *ACL*.

Terry Koo, Alexander M Rush, Michael Collins, Tommi Jaakkola, and David Sontag. 2010. Dual decomposition for parsing with non-projective head automata. In *EMNLP*. pages 1288–1298.

Adhiguna Kuncoro, Miguel Ballesteros, Lingpeng Kong, Chris Dyer, Graham Neubig, and Noah A. Smith. 2016. What do recurrent neural network grammars learn about syntax? In *CoRR*.

Siwei Lai, Liheng Xu, Kang Liu, and Jun Zhao. 2015. Recurrent convolutional neural networks for text classification. In *AAAI*. volume 333, pages 2267–2273.

Quoc V Le, Navdeep Jaitly, and Geoffrey E Hinton. 2015. A simple way to initialize recurrent networks of rectified linear units. In *CoRR*.

Geraldine Legendre, Yoshiro Miyata, and Paul Smolensky. 1990. Can connectionism contribute to syntax? harmonic grammar, with an application. In M Ziolkowski, M Noske, and K Deaton, editors, *CLS*. pages 273–252.

Erich Leo Lehmann, HJM D'Abrera, et al. 1975. *Nonparametrics*. Holden-Day.

Tao Lei, Yu Xin, Yuan Zhang, Regina Barzilay, and Tommi Jaakkola. 2014. Low-rank tensors for scoring dependency structures. In *ACL*. volume 1, pages 1381–1391.

Zhenghua Li, Min Zhang, and Wenliang Chen. 2014. Ambiguity-aware ensemble training for semi-supervised dependency parsing. In *ACL*. volume 1, pages 457–467.

Wang Ling, Tiago Luís, Luís Marujo, Ramón Fernandez Astudillo, Silvio Amir, Chris Dyer, Alan W Black, and Isabel Trancoso. 2015. Finding function in form: Compositional character models for open vocabulary word representation. In *NAACL*.

Minh-Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective approaches to attention-based neural machine translation. In *EMNLP*.

Xuezhe Ma, Zecong Hu, Jingzhou Liu, Nanyun Peng, Graham Neubig, and Eduard Hovy. 2018. Stack-pointer networks for dependency parsing. In *ACL*.

Diego Marcheggiani and Ivan Titov. 2017. Encoding sentences with graph convolutional networks for semantic role labeling. In *EMNLP*. pages 1506–1515.

Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a large annotated corpus of english: The penn treebank. *Computational linguistics* 19(2):313–330.

William Marslen-Wilson. 1973. Linguistic structure and speech shadowing at very short latencies. *Nature* 244(5417):522.

André FT Martins, Noah A Smith, Pedro MQ Aguiar, and Mário AT Figueiredo. 2011. Dual decomposition with many overlapping components. In *EMNLP*. pages 238–249.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajič. 2005. Non-projective dependency parsing using spanning tree algorithms. In *HLT/EMNLP*. pages 523–530.

Ryan T McDonald and Fernando CN Pereira. 2006. Online learning of approximate dependency parsing algorithms. In *EACL*.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *ICLR* .

Alexander H. Miller, Adam Fisch, Jesse Dodge, Amir-Hossein Karimi, Antoine Bordes, and Jason Weston. 2016. Key-value memory networks for directly reading documents. In *ACL*. pages 1400–1409.

Yusuke Miyao and Jun'ichi Tsujii. 2004. Deep linguistic analysis for the accurate identification of predicate-argument relations. In *ACL*.

Eric Moulines and Francis R Bach. 2011. Non-asymptotic analysis of stochastic approximation algorithms for machine learning. In *NeurIPS*. pages 451–459.

Joakim Nivre. 2003. An efficient algorithm for projective dependency parsing. In *The International Workshop on Parsing Technologies*.

Joakim Nivre. 2004. Incrementality in deterministic dependency parsing. In *The Workshop on Incremental Parsing*. pages 50–57.

Joakim Nivre. 2009. Non-projective dependency parsing in expected linear time. In *ACL*. volume 1, pages 351–359.

Joakim Nivre, Mitchell Abrams, Željko Agić, Lars Ahrenberg, Lene Antonsen, Maria Jesus Aranzabe, Gashaw Arutie, Masayuki Asahara, Luma Ateyah, Mohammed Attia, Aitziber Atutxa, Liesbeth Augustinus, Elena Badmaeva, Miguel Ballesteros, Esha Banerjee, Sebastian Bank, Verginica Barbu Mititelu, John Bauer, Sandra Bellato, Kepa Bengoetxea, Riyaz Ahmad Bhat, Erica Biagetti, Eckhard Bick, Rogier Blokland, Victoria Bobicev, Carl Börstell, Cristina Bosco, Gosse Bouma, Sam Bowman, Adriane Boyd, Aljoscha Burchardt, Marie Candito, Bernard Caron, Gauthier Caron, Gülşen Cebiroğlu Eryiğit, Giuseppe G. A. Celano, Savas Cetin, Fabricio Chalub, Jinho Choi, Yongseok Cho, Jayeol Chun, Silvie Cinková, Aurélie Collomb, Çağrı Çöltekin, Miriam Connor, Marine Courtin, Elizabeth Davidson, Marie-Catherine de Marneffe, Valeria de Paiva, Arantza Diaz de Ilarraza, Carly Dickerson, Peter Dirix, Kaja Dobrovoljc, Timothy Dozat, Kira Droganova, Puneet Dwivedi, Marhaba Eli, Ali Elkahky, Binyam Ephrem, Tomaž Erjavec, Aline Etienne, Richárd Farkas, Hector Fernandez Alcalde, Jennifer Foster, Cláudia Freitas, Katarína Gajdošová, Daniel Galbraith, Marcos Garcia, Moa Gärdenfors, Kim Gerdes, Filip Ginter, Iakes Goenaga, Koldo Gojenola, Memduh Gökırmak, Yoav Goldberg, Xavier Gómez Guinovart, Berta Gonzáles Saavedra, Matias Grioni, Normunds Grūzītis, Bruno Guillaume, Céline Guillot-Barbance, Nizar Habash, Jan Hajič, Jan Hajič jr., Linh Hà Mỹ, Na-Rae Han, Kim Harris, Dag Haug, Barbora Hladká, Jaroslava Hlaváčová, Florinel Hociung, Petter Hohle, Jena Hwang, Radu Ion, Elena Irimia, Tomáš Jelínek, Anders Johannsen, Fredrik Jørgensen, Hüner Kaşıkara, Sylvain Kahane, Hiroshi Kanayama, Jenna Kanerva, Tolga Kayadelen, Václava Kettnerová, Jesse

Kirchner, Natalia Kotsyba, Simon Krek, Sookyoung Kwak, Veronika Laippala, Lorenzo Lambertino, Tatiana Lando, Septina Dian Larasati, Alexei Lavrentiev, John Lee, Phng Lê H`ông, Alessandro Lenci, Saran Lertpradit, Herman Leung, Cheuk Ying Li, Josie Li, Keying Li, Kyung-Tae Lim, Nikola Ljubešić, Olga Loginova, Olga Lyashevskaya, Teresa Lynn, Vivien Macketanz, Aibek Makazhanov, Michael Mandl, Christopher Manning, Ruli Manurung, Cătălina Mărănduc, David Mareček, Katrin Marheinecke, Héctor Martínez Alonso, André Martins, Jan Mašek, Yuji Matsumoto, Ryan McDonald, Gustavo Mendonça, Niko Miekka, Anna Missilä, Cătălin Mititelu, Yusuke Miyao, Simonetta Montemagni, Amir More, Laura Moreno Romero, Shinsuke Mori, Bjartur Mortensen, Bohdan Moskalevskyi, Kadri Muischnek, Yugo Murawaki, Kaili Müürisep, Pinkey Nainwani, Juan Ignacio Navarro Horñiacek, Anna Nedoluzhko, Gunta Nešpore-Bērzkalne, Lng Nguy˜ên Thị, Huy`ên Nguy˜ên Thị Minh, Vitaly Nikolaev, Rattima Nitisaroj, Hanna Nurmi, Stina Ojala, Adédayọ Olúòkun, Mai Omura, Petya Osenova, Robert Östling, Lilja Øvrelid, Niko Partanen, Elena Pascual, Marco Passarotti, Agnieszka Patejuk, Siyao Peng, Cenel-Augusto Perez, Guy Perrier, Slav Petrov, Jussi Piitulainen, Emily Pitler, Barbara Plank, Thierry Poibeau, Martin Popel, Lauma Pretkalniņa, Sophie Prévost, Prokopis Prokopidis, Adam Przepiórkowski, Tiina Puolakainen, Sampo Pyysalo, Andriela Rääbis, Alexandre Rademaker, Loganathan Ramasamy, Taraka Rama, Carlos Ramisch, Vinit Ravishankar, Livy Real, Siva Reddy, Georg Rehm, Michael Rießler, Larissa Rinaldi, Laura Rituma, Luisa Rocha, Mykhailo Romanenko, Rudolf Rosa, Davide Rovati, Valentin Roca, Olga Rudina, Shoval Sadde, Shadi Saleh, Tanja Samardžić, Stephanie Samson, Manuela Sanguinetti, Baiba Saulīte, Yanin Sawanakunanon, Nathan Schneider, Sebastian Schuster, Djamé Seddah, Wolfgang Seeker, Mojgan Seraji, Mo Shen, Atsuko Shimada, Muh Shohibussirri, Dmitry Sichinava, Natalia Silveira, Maria Simi, Radu Simionescu, Katalin Simkó, Mária Šimková, Kiril Simov, Aaron Smith, Isabela Soares-Bastos, Antonio Stella, Milan Straka, Jana Strnadová, Alane Suhr, Umut Sulubacak, Zsolt Szántó, Dima Taji, Yuta Takahashi, Takaaki Tanaka, Isabelle Tellier, Trond Trosterud, Anna Trukhina, Reut Tsarfaty, Francis Tyers, Sumire Uematsu, Zdeňka Urešová, Larraitz Uria, Hans Uszkoreit, Sowmya Vajjala, Daniel van Niekerk, Gertjan van Noord, Viktor Varga, Veronika Vincze, Lars Wallin, Jonathan North Washington, Seyi Williams, Mats Wirén, Tsegay Woldemariam, Tak-sum Wong, Chunxiao Yan, Marat M. Yavrumyan, Zhuoran Yu, Zdeněk Žabokrtský, Amir Zeldes, Daniel Zeman, Manying Zhang, and Hanzhi Zhu. 2018. Universal dependencies 2.2. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University. http://hdl.handle.net/11234/1-2837.

Joakim Nivre, Željko Agić, Lars Ahrenberg, et al. 2017a. Universal dependencies 2.0 CoNLL 2017 shared task development and test data. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics, Charles University. http://hdl.handle.net/11234/1-2184.

Joakim Nivre, Marie-Catherine de Marneffe, Filip Ginter, Yoav Goldberg, Jan Hajič, Christopher Manning, Ryan McDonald, Slav Petrov, Sampo Pyysalo, Natalia Silveira, Reut Tsarfaty, and

Daniel Zeman. 2016. Universal Dependencies v1: A multilingual treebank collection. In *LREC*. European Language Resources Association, Portoro, Slovenia, pages 1659–1666.

Joakim Nivre, Johan Hall, and Jens Nilsson. 2006. Maltparser: A data-driven parser-generator for dependency parsing. In *LREC*. volume 6, pages 2216–2219.

Joakim Nivre, Johan Hall, Jens Nilsson, Atanas Chanev, Gülsen Eryigit, Sandra Kübler, Svetoslav Marinov, and Erwin Marsi. 2007. Maltparser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering* 13(02):95–135.

Joakim Nivre et al. 2017b. Universal Dependencies 2.0. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics, Charles University, Prague, `http://hdl.handle.net/11234/1-1983`. http://hdl.handle.net/11234/1-1983.

Stephan Oepen, Marco Kuhlmann, Yusuke Miyao, Daniel Zeman, Silvie Cinková, Dan Flickinger, Jan Hajic, and Zdenka Uresova. 2015. Semeval 2015 task 18: Broad-coverage semantic dependency parsing. In *SemEval*. pages 915–926.

Stephan Oepen, Marco Kuhlmann, Yusuke Miyao, Daniel Zeman, Dan Flickinger, Jan Hajic, Angelina Ivanova, and Yi Zhang. 2014. Semeval 2014 task 8: Broad-coverage semantic dependency parsing. In *SemEval*. pages 63–72.

Stephan Oepen and Jan Tore Lønning. 2006. Discriminant-based mrs banking. In *LREC*. pages 1250–1255.

Pāṇini and Sumitra M Katre. 1987. *Astadhyayi of Panini*. University of Texas Press.

Martha Palmer, Daniel Gildea, and Paul Kingsbury. 2005. The proposition bank: An annotated corpus of semantic roles. *Computational linguistics* 31(1):71–106.

Hao Peng, Sam Thomson, and Noah A Smith. 2017. Deep multitask learning for semantic dependency parsing. In *ACL*. volume 1, pages 2037–2048.

Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *EMNLP*.

Matthew Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. In *NAACL*. volume 1, pages 2227–2237.

Barbara Plank, Anders Søgaard, and Yoav Goldberg. 2016. Multilingual part-of-speech tagging with bidirectional long short-term memory models and auxiliary loss. *ACL* .

Carl Pollard and Ivan A Sag. 1994. *Head-driven phrase structure grammar*. University of Chicago Press.

Martin Potthast, Tim Gollub, Francisco Rangel, Paolo Rosso, Efstathios Stamatatos, and Benno Stein. 2014. Improving the reproducibility of PAN's shared tasks: Plagiarism detection, author identification, and author profiling. In Evangelos Kanoulas, Mihai Lupu, Paul Clough, Mark Sanderson, Mark Hall, Allan Hanbury, and Elaine Toms, editors, *CLEF*. Springer, Berlin Heidelberg New York, pages 268–299.

Peng Qi, Timothy Dozat, Yuhao Zhang, and Christopher D. Manning. 2018. Universal dependency parsing from scratch. In *CoNLL 2018 Shared Task*. pages 160–170. http://www.aclweb.org/anthology/K18-2016.

Peng Qi and Christopher D Manning. 2017. Arc-swift: A novel transition system for dependency parsing. In *ACL*. volume 2, pages 110–117.

Siva Reddy, Oscar Täckström, Slav Petrov, Mark Steedman, and Mirella Lapata. 2017. Universal semantic parsing. In *EMNLP*. pages 89–101.

Scott E. Reed and Nando de Freitas. 2016. Neural programmer-interpreters. In *ICLR*.

Louisa Sadler and Douglas J Arnold. 1994. Prenominal adjectives and the phrasal/lexical distinction. *Journal of linguistics* 30(01):187–226.

Ivan A Sag. 1997. English relative clause constructions. *Journal of Linguistics* 33(02):431–483.

Ivan A. Sag. 2012. Sign-based construction grammar: An informal synopsis. In Hans C. Boas and Ivan A. Sag, editors, *Sign-Based Construction Grammar*, CSLI Publications, Stanford.

Sebastian Schuster, Joakim Nivre, and Christopher D. Manning. 2018. Sentences with gapping: Parsing and reconstructing elided predicates. In *NAACL*.

Sebastian Schuster, Éric Villemonte de la Clergerie, Marie Candito, Benoît Sagot, Christopher D. Manning, and Djamé Seddah. 2017. Paris and Stanford at EPE 2017: Downstream evaluation of graph-based dependency representations. In *EPE*.

Petr Sgall, Eva Hajicová, and Jarmila Panevová. 1986. *The meaning of the sentence in its semantic and pragmatic aspects*. Springer Science & Business Media.

Peng Shi and Yue Zhang. 2017. Joint bi-affine parsing and semantic role labeling. In *International Conference on Asian Language Processing*. IEEE, pages 338–341.

Milan Straka. 2018. UDPipe 2.0 prototype at CoNLL 2018 UD shared task. In *CoNLL 2018 Shared Task*. pages 197–207. http://www.aclweb.org/anthology/K18-2020.

Milan Straka, Jan Hajič, and Jana Straková. 2016. UDPipe: trainable pipeline for processing CoNLL-U files performing tokenization, morphological analysis, POS tagging and parsing. In *LREC*. European Language Resources Association, Portoro, Slovenia.

Emma Strubell, Patrick Verga, Daniel Andor, David Weiss, and Andrew McCallum. 2018. Linguistically-informed self-attention for semantic role labeling. In *EMNLP*.

Kai Sheng Tai, Richard Socher, and Christopher D Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. In *ACL*.

Sachin S Talathi and Aniket Vartak. 2015. Improving performance of recurrent neural network with relu nonlinearity. In *ICLR*.

Duyu Tang, Bing Qin, and Ting Liu. 2015. Document modeling with gated recurrent neural network for sentiment classification. In *EMNLP*. pages 1422–1432.

Robert Endre Tarjan. 1977. Finding optimum branchings. *Networks* 7(1):25–35.

Zhiyang Teng and Yue Zhang. 2018. Two local models for neural constituent parsing. In *ACL*. pages 119–132.

Lucien Tesnière. 1959. *Eléments de syntaxe structurale*. Klincksieck.

Kristina Toutanova, Dan Klein, Christopher D Manning, and Yoram Singer. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *NAACL*. volume 1, pages 173–180.

Kristina Toutanova and Christopher D Manning. 2002. Feature selection for a rich hpsg grammar using decision trees. In *CoNLL*. volume 20, pages 1–7.

Dick C van Leijenhorst and Th P Van der Weide. 2005. A formal derivation of heaps' law. *Information Sciences* 170(2-4):263–272.

Oriol Vinyals, Łukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. 2015. Grammar as a foreign language. In *NeurIPS*. pages 2773–2781.

Hongmin Wang, Yue Zhang, GuangYong Leonard Chan, Jie Yang, and Hai Leong Chieu. 2017. Universal dependencies parsing for colloquial singaporean english. In *ACL*. volume 1, pages 1732–1744.

Sida Wang and Christopher D Manning. 2012. Baselines and bigrams: Simple, good sentiment and topic classification. In *ACL*. pages 90–94.

Yuxuan Wang, Wanxiang Che, Jiang Guo, and Ting Liu. 2018. A neural transition-based approach for semantic dependency graph parsing. In *AAAI*.

David Weiss, Chris Alberti, Michael Collins, and Slav Petrov. 2015. Structured training for neural network transition-based parsing. In *ACL*.

Hiroyasu Yamada and Yuji Matsumoto. 2003. Statistical dependency analysis with support vector machines. In *IWPT*. volume 3, pages 195–206.

Daniel H Younger. 1967. Recognition and parsing of context-free languages in time n3. *Information and control* 10(2):189–208.

Daniel Zeman, Jan Hajič, Martin Popel, Martin Potthast, Milan Straka, Filip Ginter, Joakim Nivre, and Slav Petrov. 2018. CoNLL 2018 shared task: Multilingual parsing from raw text to universal dependencies. In *CoNLL 2018 Shared Task*. pages 1–21. http://www.aclweb.org/anthology/K18-2001.

Daniel Zeman, Martin Popel, Milan Straka, Jan Hajič, Joakim Nivre, Filip Ginter, Juhani Luotolahti, Sampo Pyysalo, Slav Petrov, Martin Potthast, Francis Tyers, Elena Badmaeva, Memduh Gökırmak, Anna Nedoluzhko, Silvie Cinková, Jan Hajič jr., Jaroslava Hlaváčová, Václava Kettnerová, Zdeňka Urešová, Jenna Kanerva, Stina Ojala, Anna Missilä, Christopher Manning, Sebastian Schuster, Siva Reddy, Dima Taji, Nizar Habash, Herman Leung, Marie-Catherine de Marneffe, Manuela Sanguinetti, Maria Simi, Hiroshi Kanayama, Valeria de Paiva, Kira Droganova, Hěctor Martínez Alonso, Hans Uszkoreit, Vivien Macketanz, Aljoscha Burchardt, Kim Harris, Katrin Marheinecke, Georg Rehm, Tolga Kayadelen, Mohammed Attia, Ali Elkahky, Zhuoran Yu, Emily Pitler, Saran Lertpradit, Michael Mandl, Jesse Kirchner, Hector Fernandez Alcalde, Jana Strnadova, Esha Banerjee, Ruli Manurung, Antonio Stella, Atsuko Shimada, Sookyoung Kwak, Gustavo Mendonça, Tatiana Lando, Rattima Nitisaroj, and Josie Li. 2017. CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies. In *CoNLL 2017 Shared Task*. Association for Computational Linguistics.

Rui Zhang, Cicero Nogueira dos Santos, Michihiro Yasunaga, Bing Xiang, and Dragomir Radev. 2018a. Neural coreference resolution with deep biaffine attention by joint mention detection and mention clustering. *arXiv preprint arXiv:1805.04893* .

Xingxing Zhang, Jianpeng Cheng, and Mirella Lapata. 2017. Dependency parsing as head selection. *EACL* .

Yu Zhang, Guoguo Chen, Dong Yu, Kaisheng Yaco, Sanjeev Khudanpur, and James Glass. 2016. Highway long short-term memory RNNs for distant speech recognition. In *ICASSP*. IEEE, pages 5755–5759.

Yuan Zhang and David Weiss. 2016. Stack-propagation: Improved representation learning for syntax. In *ACL*. volume 1, pages 1557–1566.

Yuhao Zhang, Peng Qi, and Christopher D Manning. 2018b. Graph convolution over pruned dependency trees improves relation extraction. In *ACL*.

GK Zipf. 1949. *Human Behaviour and the Principle of Least-Effort*. Addison-Wesley.