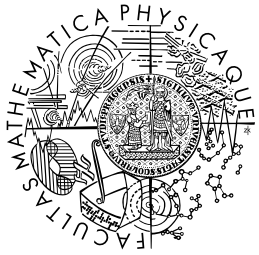

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Vinit Ravishankar

Parsing of Texts with Code-Switching

Institute of Formal and Applied Linguistics

Supervisor of the master thesis:	RNDr. Daniel Zeman, Ph.D.
Study programme:	Computer Science
Study branch:	Computational Linguistics

Prague 2018

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague date 20/07/2018 Vinit Ravishankar

This thesis would not have been possible without the support of my family and friends, and it would have been a lot worse without fruitful discussion with Mostafa and Artur – the MGAD lads – whom I learnt a lot from and had a great time with. I would also like to thank Memduh and Fran for their top-notch linguistic and political banter (and advice) that always kept me motivated. I am also very grateful to my supervisors, Dr. Daniel Zeman and Dr. Albert Gatt, who always had valuable advice and feedback for me.

My master's study has largely been funded by the Erasmus Mundus Language and Communication Technologies grant, without which I would be infinitely more broke than I am.

Title: Parsing of Texts with Code-Switching

Author: Vinit Ravishankar

Institute: Institute of Formal and Applied Linguistics

Supervisor: RNDr. Daniel Zeman, Ph.D., Institute of Formal and Applied Linguistics

Abstract: The aim of this thesis is twofold; first, we attempt to dependency parse existing code-switched corpora, solely by training on monolingual dependency treebanks. In an attempt to do so, we design a dependency parser and experiment with a variety of methods to improve upon the baseline established by raw training on monolingual treebanks: these methods range from treebank modification to network modification. On this task, we obtain state-of-the-art results for most evaluation criteria on the task for our evaluation language pairs: Hindi/English and Komi/Russian. We beat our own baselines by a significant margin, whilst simultaneously beating most scores on similar tasks in the literature. The second part of the thesis involves introducing the relatively understudied task of predicting code-switching points in a monolingual utterance; we provide several architectures that attempt to do so, and provide one of them as our baseline, in the hopes that it should continue as a state-of-the-art in future tasks.

Keywords: parsing dependency parsing treebank universal dependencies code switching

Contents

Introduction **3**

1 Prior Work **6**
1.1 Dependency Annotation . . . 6
1.1.1 Universal Dependencies . . . 6
1.1.2 'Universal' features . . . 7
1.2 Dependency Parsing . . . 8
1.2.1 Transition- vs graph-based approaches . . . 8
1.2.2 Statistical vs neural approaches . . . 9
1.2.3 Evaluation . . . 9
1.3 Code switching . . . 10
1.3.1 Description . . . 10
1.4 Computational processing of code-switching . . . 10
1.4.1 Resources . . . 10
1.4.2 Methods . . . 12

2 Dependency Parsing **14**
2.1 Artificial neural networks . . . 14
2.1.1 Artifical neurons . . . 14
2.1.2 Forward propagation . . . 14
2.1.3 Backpropagation . . . 15
2.2 Embeddings and representations . . . 17
2.2.1 Task-specific embeddings . . . 18
2.2.2 Pretrained embeddings . . . 19
2.3 Recurrent neural networks . . . 19
2.3.1 Naive RNNs . . . 19
2.3.2 Vanishing gradients, and onward . . . 20
2.3.3 Long short-term memory . . . 21
2.3.4 Character-level RNNs . . . 23
2.4 Multi-task learning . . . 24
2.5 Dependency parser . . . 26
2.5.1 Architecture . . . 26
2.5.2 Implementation . . . 28
2.5.3 Hyperparameters and optimisation . . . 29

3 Parsing for Code-Switched Languages **30**
3.1 Data and baselines . . . 31
3.1.1 Treebanks . . . 31
3.1.2 Code-switching statistics . . . 31
3.1.3 Constructing a baseline . . . 33
3.2 Word representations . . . 35
3.2.1 Mapped embeddings . . . 35
3.2.2 Cross-lingual dependency parsing . . . 36
3.2.3 Embeddings and methods . . . 37
3.3 Treebank-level modifications . . . 40

3.3.1 Algorithms . . . 41
3.3.2 Evaluation . . . 42
3.4 Network alterations . . . 44
3.4.1 Language ID . . . 44
3.4.2 Multi-task learning . . . 47
3.4.3 Domain shift . . . 50
3.4.4 Development weight learning . . . 53

4 Predicting code-switch points **57**
4.1 Background . . . 57
4.2 Evaluation . . . 57
4.2.1 Motivation . . . 57
4.3 Pre-processing and data . . . 58
4.4 Evaluation . . . 59
4.5 Analysis . . . 60

Conclusion **63**

Bibliography **67**

List of Figures **76**

List of Tables **77**

Introduction

This thesis primarily presents a thorough investigation into the phenomenon of code-switching, and the relevance of phenomenon to the task of dependency parsing. The nature of this research makes it rather cross-domain; as such, we emphasise both the crucial linguistic phenomena that are apparent through code-switching, whilst simultaneously providing a rigid mathematical and computational analysis of the systems we designed to deal with the task.

Whilst both dependency parsing and code-switching are tasks, or phenomena, that have been studied by computer scientists and by linguists alike, their intersection is a fairly modern phenomenon. We posit that this is due to several factors. One of these is undoubtedly the rapidly increasing relevance of dependency parsing as a rigid, cross-linguistically valid downstream NLP task. This is, in part, due to the success of the Universal Dependencies project [Nivre et al., 2016] - a multilingual, yet cross-linguistically consistent dependency treebank annotation project. Universal Dependencies (also referred to with the abbreviation 'UD'), consists of a large number of consistently annotated treebanks: 122 treebanks in 71 languages, as of v2.2. Many attempts to parse these treebanks have been made, ranging from transition- to graph-based, and from statistical to neural. These attempts are often the result of shared tasks, the first (and most recent) of which is the CoNLL shared task on dependency parsing [Zeman et al., 2017], held in 2017.

We defer more detailed explanations of dependency grammars, and dependency parsing, to a later section: however, a brief definition is that dependency grammars are an annotation system that, fundamentally, attempt to capture *relations* between words in a sentence, rather than recursively decompose a sentence into its constituent chunks, as in constituency parsing. Dependency parsing is, therefore, the task that "solves" dependency grammars: it involves learning two elements from gold-standard data: specifically, the dependency tree structure - which is fundamentally a connected acyclic graph, where every word represents a node in the graph, and arcs in the graph represent relations between the words - and the nature of these relations, specifically annotated according to UD standards.

There are two major approaches to dependency parsing: transition- and graph-based parsing. The former (predominantly; minor variations do exist) attempts to represent the building of a tree as a sequence of transitions, and learn a set of operations to apply iteratively to a stack and a buffer, until a valid parse tree is obtained. The algorithm, therefore, never truly learns how a tree looks: it learns how to build one from a series of transitions that assign arcs.

Graph-based parsing, on the other hand, attempts to directly predict a graph structure that may or may not obey tree constraints: later, these graphs are reduced to their minimum (or, rather, expressed as log probabilities, maximum) spanning trees [Chu and Liu, 1965b, Edmonds, 1967].

We go into more detail on dependency parsing and our approach to this task in a later section.

Code-switching is a fascinating linguistic phenomenon that has been extensively studied, historically and today, albeit primarily by linguists [Auer, 2013,

Milroy and Muysken, 1995]. Whilst all-inclusive, simple definitions of this phenomenon do not strictly exist, code-switching is, *in most situations*, a phenomenon where multilingual speakers generate utterances that include components from all or some of their languages. Components, in this context, does not necessarily mean words: it can also refer to clauses, or even sentences. This phenomenon is visible in different forms in large parts of the world.

Computational processing of code-switched language has also picked up in recent years; this has touched upon several downstream tasks, such as POS tagging, though many remain as yet unstudied, or relatively understudied. One of the most interesting aspects of this computational processing is the question of the necessity of corpora. Whilst unsupervised techniques for most NLP tasks have existed for years, these rarely, if ever, outperform supervised tasks, making the existence of an annotated corpus essential. Whether, and to what degree, these corpora are essential for code-switched languages is a complex issue, and a large part of our research question.

Research goals

The combination of these two distinct themes leads to several research goals, which we attempt to summarise and enumerate:

1. Given annotated monolingual corpora in the two (or more) 'constituent' languages that make up a code-switched treebank, what techniques can be leveraged to best utilise these, and to obtain what sorts of results?

2. Can models trained on these multiple monolingual corpora, if sufficiently well-resourced, outperform models that were trained using state-of-the-art monolingual techniques on relevant code-switched corpora? To what extent is this true?

3. And, finally, can computers learn to predict, given code-switched sentences, when a switch is going to occur? i.e., can they model human switching patterns and behaviour?

Our research touches upon all four of these questions; however, we focus primarily on the first, which is on creating techniques that specifically improve results on code-switched tasks.

Structure

This thesis is divided into several chapters, each with a separate, specific focus, that ties to our research questions.

I Prior Work

In this chapter, we provide a thorough background to all the numerous constituent elements of this thesis; both from a linguistic and a computational perspective, as well as a review of prior work on computational processing of code-switched language, that may or may not be relevant specifically to the problem of dependency parsing.

II Dependency Parsing

This section provides a solid background on neural networks, as relevant to our task; it also provides a background on dependency parsing, including the pragmatics behind our implementation of a dependency parser.

III Parsing for Code-Switched Languages

This section, as the 'main' part of this thesis, elaborates on the different methods we follow in an attempt to optimise our parser's performance on code-switched data, in the absence of relevant code-switched training data.

IV Code-switching Point Detection

Whilst not the primary focus of this thesis, this section outlines the task of detecting code-switching points in text streams. It also highlights some of the possible uses of this task, and provides a thorough evaluation of multiple neural architectures to a task that has not been revisited in the recent past.

Future Work and Conclusions

In this section, we analyse our experiments from the previous chapters, and discuss their implications, both on parsing and on code-switched data. We also discuss possible future avenues for further research.

1. Prior Work

1.1 Dependency Annotation

Dependency syntax annotation has existed in some form or other for centuries [Percival, 1990]. Put very simply, dependency parsing is opposed to constituency parsing, in that it is a set of formalisms for parsing sentences that, rather than recursively decompose sentences into phrase-based structures, instead directly annotate relations between words. Nivre provide an excellent review on the discipline, along with relevant history.

For some brief background on dependency annotation relevant to our work, we provide an overview of the largest and most well-known efforts in dependency annotation, and their relevance to UD.

One of UD's direct ancestors, the Stanford Typed Dependencies representation for English [de Marneffe and Manning, 2008], was a large-scale effort that attempted dependency parsing for English, as a resource for Stanford's textual entailment systems; this system eventually began to be adapted to other languages. Independent annotation projects for other languages also did exist; notable efforts include, for instance, the comprehensive Prague Dependency Treebank [Hajič et al., 2017] for Czech, and SynTagRus [Boguslavsky et al., 2000] for Russian. Other resources relevant to the establishment of UD as a standard include the original Google Universal dependencies treebank [McDonald et al., 2013], Google's universal part-of-speech tagset [Petrov et al., 2011], and, crucially, the Interset tagset conversion system [Zeman, 2008], for enabling reusable tagset conversion utilities.

Universal Dependencies originated as a means to unify all these various strands of annotation under a cross-linguistically consistent umbrella. It has seen multiple releases, each more comprehensive than the last: our data follows the UDv2.0 syntactic annotation guidelines.

1.1.1 Universal Dependencies

In this section, we provide a brief description of the CoNLL-U format used in the Universal Dependencies schema. CoNLL-U files fundamentally consist of ten-column lines, each line denoting a word: every column serves to indicate a specific feature of the word.

1. **ID**: the word index within the sentence; begins at 1. Multiword tokens are expressed via ranges.
2. **FORM**: the surface form of the word as it appears in running text.
3. **LEMMA**: the lemmatised or stemmed form of the word; lemmatisation standards vary across treebanks.
4. **UPOS**: the *universal* part-of-speech tag of the word.
5. **XPOS**: the *language-specific* part-of-speech tag of the word; these are defined per language, with underscores used if they are not available.

	Nominals	Clauses	Modifier words	Function Words
Core arguments	nsubj obj iobj	csubj ccomp xcomp		
Non-core dependents	obl vocative expl dislocated	advcl	advmod* discourse	aux cop mark
Nominal dependents	nmod appos nummod	acl	amod	det clf case
Coordination	**MWE**	**Loose**	**Special**	**Other**
conj cc	fixed flat compound	list parataxis	orphan goeswith reparandum	punct root dep

Figure 1.1: Universal dependency relations as of v2.0; source `universaldependencies.org/u/dep/`

6. **FEATS**: the morphological features of the word; the morphology is drawn from the universal set of morphological features, or language-specific extensions.
7. **HEAD**: the index of the head of the current word; 0 if the word is the root.
8. **DEPREL**: the specific universal dependency relation between the word and its head. Optional subtypes for specific languages are allowed.
9. **DEPS**: optional *enhanced* dependency relations with one or more other tokens in the sentence. These are absent from most treebanks.
10. **MISC**: miscalleneous information, represented with key-value pairs. This is exceptionally useful to us, as we represent language in this field.

Each sentence is expected to begin with a raw, untokenised version of the sentence as a comment, preceded by a `#`, along with metadata information for that sentence. Sentences are separated by blank lines.

1.1.2 'Universal' features

As was mentioned in the previous section, Universal Dependencies includes a set of POS tags (UPOS), a set of morphological key/value pairs (FEATS) and a set of dependency relations (DEPREL) that are considered 'universal', i.e. they are expected to apply to all languages within UD.

These include a set of universal features, described in Figure 1.1. Each feature can be optionally subtyped for a specific language, however, this is avoided as far as possible, to retain the 'universalness' of the feature space. Constraining the size of the feature space also allows for easier parsing.

1.2 Dependency Parsing

Fairly obviously, dependency parsing is the task of predicting, given a sentence and some variable amount of training data, the valid dependency parse for that particular sentence. Whilst modern systems require parsers to produce *the* correct parse, according to some human-annotated evaluation set, for a parse to be considered correct, proposals to relax these constraints have been made [Søgaard, 2017]. In this section, we provide a quick overview of some of the different paradigms in dependency parsing, although we would like to mention that this is by no means comprehensive: we leave out several sorts of parsers (such as, for instance, rule-based parsers).

1.2.1 Transition- vs graph-based approaches

Transition-based parsing

Transition-based dependency parsing has long been the predominant method used to produce dependency parses. It has its roots in shift-based parsing, used for parsing programming languages [Aho and Ullman, 1972]. Transition-based parsing systems, rather than attempting to directly produce parses, instead rely on reducing parses to sequences of *transitions*. A parsing system then attempts to learn how to produce a sequence of transitions, rather than how to immediately predict a tree.

There are a variety of 'transition systems' in use, the most well-known (and simplest) of which is Nivre's arc-standard parsing model, which uses three transitions: namely, `shift`, `left_arc` and `right_arc`. The first of these merely moves a token from the stack to the buffer; the other two assign a parent to a particular node, with the transition determining the direction. One of its obvious flaws is its inability to deal with non-projectivity, or crossing arcs, within a sentence; there have been numerous proposed fixes for this, such as introducing newer transitions such as the `swap` transition.

The sequence-prediction fundamentals give transition-based parsing an instant advantage: it can guarantee a well-formed tree in the output.

Graph-based parsing

Graph-based parsers are a relatively new parsing method; unlike transition parsing, graph parsers require significantly more complex features. The advent of neural networks and their ability to model complex feature spaces in arcs has led to a resurgence in the popularity of graph parsing systems, that are, conceptually, much simpler than transition-based ones.

Fundamentally, graph-based parsers attempt to learn the arc probability between every pair of words in the sentence, essentially generating a fully-connected graph, with edges weighted by arc probabilities. If a dependency 'tree' is unnecessary - which can be the case for certain downstream applications - the highest weighted edges are naively selected. If, however, a tree constraint is necessary, the minimum (based on distances; maximum based on weights) spanning tree of the graph is obtained, giving us the optimal parse tree.

1.2.2 Statistical vs neural approaches

Statistical parsing

Statistical dependency parsing is also, often, referred to as 'data-driven dependency parsing': this is not to imply that neural parsing models do not use data, but was a term used more to distinguish statistical models from rule-based ones (that are beyond the scope of this thesis). Statistical dependency parsers use complex, often human-defined, feature functions, that attempt to extract information from either tokens, or transitions - depending on the specific model of parsing being used - to supply to a learning system that attempts to align features to parses. One of the more famous examples of traditional data-drives parsers is MaltParser [Nivre et al., 2007], which used files consisting of complex feature definitions - such as, for instance, the 'child of the second element on the buffer'. The use of statistical parsers has been on the wane lately, with the advent of neural parsing systems.

Neural parsing

The rapid adoption of neural networks by the NLP community in recent years, owing largely to their success in fields like image recognition [Lopez and Kalita, 2017] is also a phenomenon that is visible in dependency parsing. A variety of approaches to parsing using neural networks have been attempted, ranging from convolution [Zhang et al., 2016] to reinforcement learning-based networks [Zhang and Chan, 2009].

In principle, however, every method relies on reducing informative elements – such as tokens, or POS tags, either as inputs or as stack/buffer elements – to some form of numeric representation, typically dense vectors. These vectors are then embedded into some form of neural architecture that is capable of learning to produce labelled dependency parses from them.

We provide more background on the principles of neural networks, and their application to dependency parsing, in Chapter 2.

1.2.3 Evaluation

Dependency parsers are, in a modern setting, generally evaluated on the basis of two solid metrics - the unlabelled attachment score (that we refer to as the 'UAS'), and the labelled attachment score ('LAS'). Fundamentally, the UAS is the proportion of dependency arcs that our parser successfully managed to predict, whilst the LAS is the proportion of arcs that our parser successfully predicts, along with the labels associated with those arcs [Buchholz and Marsi, 2006].

Most parser evaluation is, today, conducted within the constraints of the Universal Dependencies project; comprehensive efforts at parser evaluation include the highly successful series of CoNLL shared tasks on dependency parsing [Zeman et al., 2017], where multiple parsers are rigidly evaluated against most UD treebanks. Other variants on the standard UAS/LAS metric have been introduced, such as the weighted LAS (wLAS) – which downsamples the contribution of 'easy' dependency relations (like punctuation) to the final score – or the morphologically aware LAS (mLAS) – which weights predictions based on their performance

on morphological analysis. Throughout this thesis, we use UAS, LAS and wLAS as our key evaluation metrics.

1.3 Code switching

1.3.1 Description

Code-switching is a fairly common, yet computationally relatively understudied, linguistic phenomenon, that occurs predominantly due to some form of language contact. Fundamentally, it is described by multilingual speakers switching between two or more languages *at least* within a single conversation. Code-switching is, understandably, a very hazy phenomenon that is hard to accurately describe in a way that covers all social contexts in which it occurs.

Several ways of quantifying code-switching exist, one such method including different levels of code-switching. For instance, code-switching at a token level is a fairly common phenomenon (albeit hard to discern from lexical borrowing; see Myers-Scotton [1992]). Partly due to globalisation and the Internet, a lot of languages that lack historical ties to the English language have begun displaying code-switching with English tokens, such as German [Zhiganova, 2016] and Dutch [Dongen, 2017].

Apart from the modern use of internet slang across languages, code-switching is particularly visible in a post-colonial context [Ferguson, 2003]. This is a phenomenon common enough in many former British colonies, such as the Indian subcontinent, where Hindi/English code-switching is often referred to as *'Hinglish'* [Sailaja, 2011, Si, 2011], or in Malta, with *'Maltenglish'* [Francesconi, 2010, Camilleri, 1996]. These language often also, for instance, exhibit code-switching at a clausal or morphological level.

1.4 Computational processing of code-switching

Despite code-switching being a well-studied field, applications of insights from NLP to parse code-switched data are relatively fairly limited. We hypothesise that this is largely due to the lack of relevant data, which, in part, is a result of economic conditions, as well as due to code-switching often being looked down on by language purists. In this section, we provide an overview of work done on computational processing of code-switched language, particularly relevant to dependency parsing, and how we intend to build on this work.

1.4.1 Resources

Unannotated corpora

The gathering and annotation of corpora is a laborious task that involves a significant amount of human effort. There exist, however, several - fairly recent - corpora, for code-switched language. Unfortunately, most of these corpora are merely raw text streams, without any sort of annotation - however, the variety and availability of these corpora is rapidly growing.

Unannotated code-switched corpora exist for several language "pairs". One of the earliest efforts is the creation of an Hindi/English code-switching corpus [Dey and Fung, 2014], which involved the creation and transcription of student 'interviews' on informal themes. Lyu et al. [2015] describe a Mandarin-English code-switched corpus, which was also built on recordings and transcriptions of unscripted conversations. Hamed et al. [2018] provide a corpus of Egyptian Arabic-English data, also obtained through interviews.

There also exist pairs that do not include English as one of the languages: for instance, Cotterell et al. [2014] provide a transcribed corpus of Arabic-French code-switching, consisting mainly of newspaper text. Relevant to this thesis is a corpus of spoken Komi-Russian[1] [Partanen et al., 2018]; another relevant corpus is an annotated German-Turkish corpus [Çetinoğlu, 2016].

The diversity of reasons for the existence of code-switching in all the above pairs is quite significant: ranging from colonialism to globalism to immigration, to mixtures of the three to varying extents.

Annotated corpora

Creating annotated corpora is significantly more effort than merely assembling multilingual corpora, as it requires significantly more human effort. Despite this, there exist several annotated corpora for a variety of tasks, the most common of which is (understandably) part-of-speech (or POS) tagging [Vyas et al., 2014, Solorio and Liu, 2008b, Çetinoğlu and Çöltekin, 2016, Nelakuditi et al., 2016]. POS tagging is a fairly 'basic' downstream NLP task, that involves predicting annotations of tokens with specific parts of speech, according to a certain schema. The task is, therefore, a fairly simple sequence labelling task.

Extremely relevant to our task are corpora that have been annotated specifically for dependency parsing. These, specifically, include a Hindi/English code-mixed treebank [Bhat et al., 2017], and a Komi-Russian one [Partanen et al., 2018]. Neither of these treebanks is what could be considered *large*: the treebank for Hindi/English consists of a total of 6,789 tokens (divided into development and test sets of 3,467 and 3,322 tokens respectively). Komi-Russian datasets are even smaller: 105 sentences, 80 of which are adapted variants of a monolingual corpus to try and reflect code-switching that would occur in normal conversations. Given that the minimum treebank size for inclusion into the CoNLL-2017 shared task [Zeman et al., 2017] was a total of 10,000 tokens, it is clear that neither of our code-switched treebanks is particularly large.

It is crucial to mention, at this point, the existence of a substantially larger Hindi/English code-switched corpus [Bhat et al., 2018]; one that includes sufficient data for a training set. We do *not* use this dataset in our experiments, as the whole point of our work is to evaluate techniques for improving parsing performance in the absence of much code-switched data; however, we use the results presented in this work as an 'upper baseline', as it were: one that we hope, but do not expect, to beat.

[1] Komi is a minority Uralic language spoken in the Komi Republic, in Russia

1.4.2 Methods

There has been a not-insignificant amount of work on computational processing of code-switched language in recent years, particularly with Indian languages and English. A variety of tasks have been studied, and we attempt, in this section, to outline the work done on some of them. For brevity, we expect the reader to be broadly familiar with each of the following tasks in a *monolingual* setting, and provide a review of the literature as relevant to code-switching.

POS tagging

Part-of-speech tagging, or POS tagging, is a task that can absolutely be considered a 'core' NLP task. There have been several POS taggers that were specifically either evaluated on code-switched data, or specifically adapted for code-switched data. Ghosh et al. [2016] describe a CRF-based POS tagger for code-switched data in Bengali, Hindi and Tamil, with English; this was submitted to a shared task on parsing code-switched Indian languages. Whilst their results are significantly above the baseline, they also have access to annotated code-switched data for training, as do other competitors in the shared task(s). AlGhamdi et al. [2016] provide several more interesting architectures, evaluated on Spanish/English, and Modern Standard Arabic (MSA)/dialectal Arabic. Their first architecture attempts language identification first, then uses two monolingual taggers to tag the entire sentence: each tagger tags a relevant chunk for its particular language. Their second model obtains two tag sequences over complete sentences, by applying either monolingual tagger to the whole sentence: it then attempts language identification, and then interpolates the language data with the monolingual tag sequences to obtain the relevant language tag. Their third system uses confidence scores output by each tagger, for each tag, to choose the tag that has higher confidence associated with it. Finally, their last system teaches a classifier (specifically, an SVM) to choose which tag to output.

One of the most widely cited works on POS tagging code-mixed text is Solorio and Liu [2008b], for English/Spanish; an interesting contribution they make is the use of hand-crafted heuristics, such as making use of the 'foreign' POS tag output by their monolingual English or Spanish taggers, and forcing their system to output the opposite language tag if one of the two languages results in a 'foreign' POS tag. Patel et al. [2016] provide a more modern approach to POS tagging, and use character-level RNNs: however, they, too, have access to code-switched training data. Finally, Bhat et al. [2018] train a tagger[2] jointly with their parser, using character-level RNNs: however, they also use their parsing loss to train the shared tagger and parser layers. These sorts of architectures are explained in greater detail in Section 2.4.

Language identification

Language identification, which is (rather obviously) the task of identifying the language of each token in a sentence, is a task that has seen excellent results, with the exception of scenarios where code-switching is across languages that are

[2]Note that this paper was made public towards the end of this thesis being written.

extremely similar, such as modern standard Arabic (MSA) and dialectical Arabic (DA). Rather conveniently, Molina et al. [2016] provide a concise overview for their second shared task on language identification in code-switched language: the languages the shared task focussed on are MSA/DA and Spanish/English. Their data sets for both languages are drawn from Twitter. Whilst all the submitted systems beat their baseline for Spanish/English by a significant margin, the best-performing system [Shirvani et al., 2016] used logistic regression and several external resources, such as named entity recognition systems. The next best performing system, which also happened to be the best performing system for MSA/DA [Samih et al., 2016], used recurrent neural networks (specifically LSTMs) for language identification. Finally, whilst by no means the best-performing system on either pair, Jaech et al. [2016] provide a fascinating language identification model using convolutional neural networks.

Dependency parsing

To the best of our knowledge, there exist, as of now, only two works on dependency parsing code-switched language, both of which were written during the compilation of this thesis. Whilst we have referenced both these papers in 1.4.1, here, we describe the methods they have used, rather than their datasets.

The first of the two, Partanen et al. [2018], fundamentally relies on modifying the dependency parsing architecture described by Lim and Poibeau [2017] to include multilingual word representations; they then provide this system with mapped embeddings, à la Artetxe et al. [2016], and attempt to parse their Komi/Russian code-mixed corpus.

More comprehensively, Bhat et al. [2018] provide an end-to-end system for segmentation, language identification, text normalisation, POS tagging and dependency parsing. In doing so, they use a transition-based parser, where their POS tagger and parser act as stacked components.

It is important, here, for us to constrain the aim of this thesis: whilst an end-to-end system is, indeed, an extremely useful system, we choose to focus solely on the task of dependency parsing. Our rationale behind this is manifold:

1. There has been significant enough research into other tasks on code-switched data that it would take significant amounts of effort to beat existing states-of-the-art.

2. An end-to-end system can, in theory, be constructed by simply stacking individual states-of-the-art[3]; there is nothing (again, in theory) that would stop one from doing so with our parser.

3. The relative lack of focus on dependency parsing as a task implies that we can realistically run several experiments that, even if unsuccessful, would be informative for future research as a negative paper.

[3]Although recent innovations in the domain of multi-task learning show that such systems may not prove to be the most optimal ones

2. Dependency Parsing

In this section, we describe the broad architecture of our dependency parser. The architecture of our parser broadly follows the architecture followed by Dozat and Manning [2016], with several changes: and, indeed, modifications, in an attempt to capture code-switching well. Our parser predominantly uses neural networks to learn to parse; we therefore preface this section with a large introduction to neural networks and their specific use in parsing.

2.1 Artificial neural networks

Artificial neural networks, despite their broad acceptance by the NLP community being less than a decade old, are, conceptually, fairly old. Initial research in the application of neural networks to computing (and not biology) date back to the 1940s [McCulloch and Pitts, 1943]. Major breakthroughs include the design of the perceptron [Rosenblatt, 1958], a simple addition/subtraction based model. Significant improvements in the feasibility of neural networks were realised by the development of the backpropagation algorithm [Rumelhart et al., 1985].

In recent years, neural networks have rapidly gained acceptance largely due to their new-found tractability, thanks to advances in parallel computing and GPUs; the most significant recent work, that firmly established the use of neural networks in image recognition, was the deep convolutional network AlexNet [Krizhevsky et al., 2012].

It is important to note that a lot of what we know of neural networks was built standing on the shoulders of giants; significant amounts of insight from classical machine learning are still extremely valuable.

2.1.1 Artifical neurons

Fundamental to artificial neural networks is the 'neuron'; despite the arcane name, inspired by neuroscience and the structure of the brain, a neuron is fundamentally a 'node' that composes multiple inputs into a single output, typically with the neuron introducing a non-linearity.

Composition at a neuron merely involves multiplying an input with a weight (and optionally adding a bias) associated with the input and that particular neuron: the output is then transformed by a non-linear function.

Figure 2.1 demonstrates the transformation of four inputs by a neuron (without biases): each input is multiplied by its corresponding weight at that neuron, and a non-linear function f is applied to their linear combination. It is immediately obvious why f is necessary in deep networks: series of linear operations on input data can trivially be composed to single linear operations, making non-linearities essential.

2.1.2 Forward propagation

Typically, neural networks consist of a series of neurons composed into a single *layer*, and multiple such layers (hence the term 'deep'). This, fundamentally,

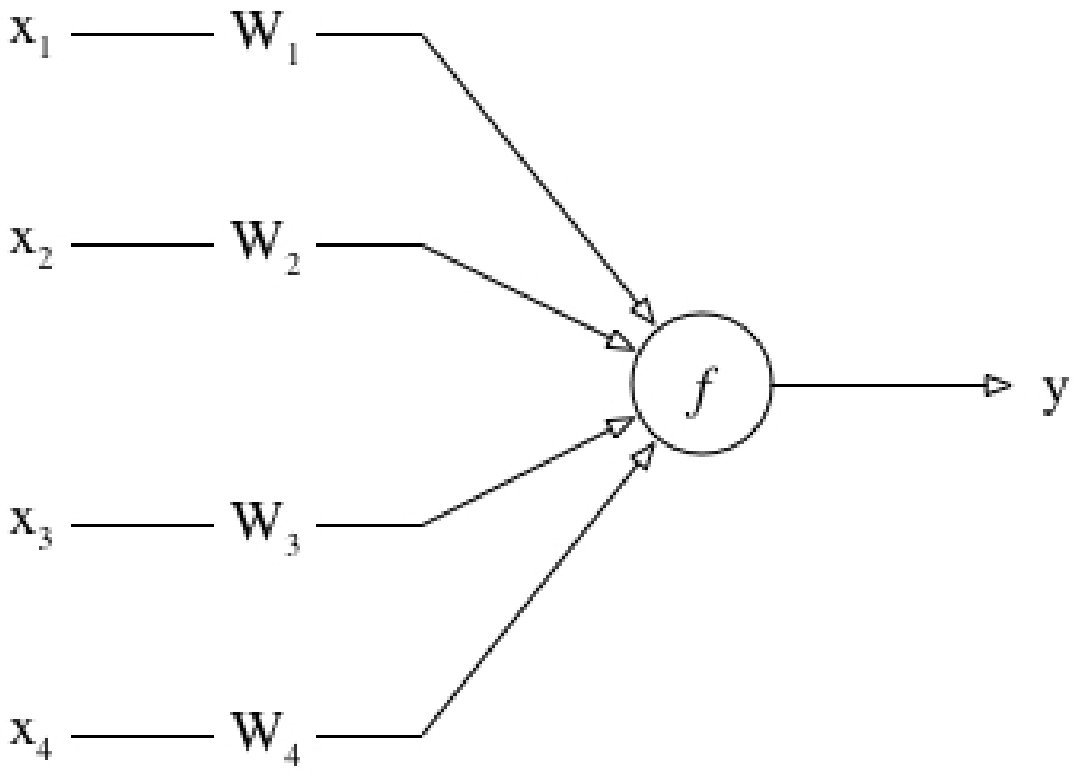

Figure 2.1: A basic example of a neuron with four inputs

results in a series of non-linear transformations to the input data.

Figure 2.2 is a fully connected neural network with a single hidden layer with three neurons: each neuron in the hidden layer applies a non-linear transformation to a weighted combination of the four inputs, x_1 and x_2. The outputs of these neurons are then recombined to form a single output, y.

This, fundamentally, is the 'forward propagation' step: weights and biases are first initialised randomly, usually following some sort of normal distribution [Glorot and Bengio, 2010]. Inputs are then transformed using these weights and biases, and an output is obtained. The 'loss' of this output relative to the gold-standard output is calculated, and weights and biases are reset by backpropagating these losses.

2.1.3 Backpropagation

Fundamental to the design of neural networks is the principle of backpropagation, which is, essentially, the standard way neural networks learn to correct their weights and biases. Backpropagation fundamentally involves using a variant of gradient descent to optimise weights and biases. Past the forward propagation step, there needs to be a way to correct, or nudge, the weights and biases forward to get a better representation.

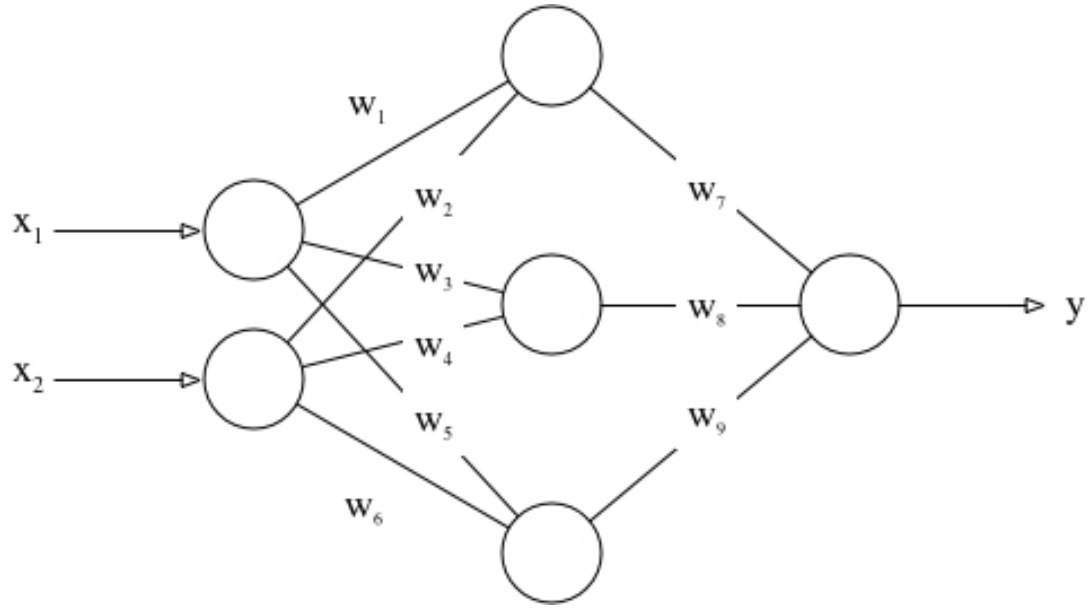

Figure 2.2: A neural network with one hidden layer and two inputs

Gradient descent

In order to understand why backpropagation works the way it does, it is, first, crucial to understand gradient descent. Given a loss function L, and a set of parameters[1] θ for which the loss is to be calculated, i.e. a function that returns how 'off' a prediction is compared to a gold standard, it is obvious that $L(\theta)$ has its minima where $L'(\theta) = 0$. Gradient descent, in its most simplified and naive form, involves subtracting the slope of $L(\theta)$ from θ: the slope, here, being, fundamentally, a signed value indicating the magnitude and direction of the gradient at that point. This subtraction is typically scaled by a 'learning rate' α. Thus, with straightforward, naive gradient descent, we can reset our parameters (using the vector derivative operator ∇) as:

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta)$$

Calculating derivatives

It is clear from the previous section that our parameters need to be adjusted based on the value of the slope of some loss L, relative to the specific parameter, generalised to all parameters with θ. The chain rule from elementary calculus proves exceptionally useful in calculating these derivatives.

Consider, for instance, the same simple single-hidden-layer network in Figure 2.2; let us assume all neurons are unbiased, for simplicity. It is clear that we need to calculate several weight parameters: the weights between every input and every hidden neuron, and every hidden neuron (with activation function f) and the single output. Let the weighted combination of inputs at a hidden neuron n be represented by i_n, and the output of the non-linearity applied at that neuron by h_n. Let us assume our loss is a simple mean-squared error (MSE) loss:

[1] 'Parameters' merely refers to the weights and biases etc. used in a network

$$L = (y_{gold} - y)^2$$

Assuming we need to calculate the derivative of the loss w.r.t. weight w_1, we have:

$$\begin{aligned}\frac{\partial L}{\partial w_1} &= \frac{\partial L}{\partial y}\frac{\partial y}{\partial w_1} \\ &= \frac{\partial L}{\partial y}\frac{\partial y}{\partial h_1}\frac{\partial h_1}{\partial w_1} \\ &= \frac{\partial L}{\partial y}\frac{\partial y}{\partial h_1}\frac{\partial h_1}{\partial i_1}\frac{\partial i_1}{\partial w_1}\end{aligned}$$

All these *individual* derivatives are easy to calculate:

$$\begin{aligned}\frac{\partial L}{\partial y} &= 2(y_{gold} - y) \\ \frac{\partial y}{\partial h_1} &= \frac{\partial}{\partial h_1} h_1 w_7 = w_7 \\ \frac{\partial h_1}{\partial i_1} &= f'(i_1) \\ \frac{\partial i_1}{\partial w_1} &= x_1\end{aligned}$$

And our equation thus composes to:

$$\frac{\partial L}{\partial w_1} = 2(y_{gold} - y) \cdot w_7 \cdot f'(i_1) \cdot x_1$$

the value of which is then subtracted from w_1.

This series of forward- and backpropagation is computed for every input element; typically, the inputs are grouped into 'batches', to prevent erratic fluctuations in gradients. The operations are then, typically, repeated over the same input data: each iteration of forward propagation followed by parameter recalculation via backpropagation is called an *epoch*.

2.2 Embeddings and representations

Fundamental to the use of neural networks for most NLP applications is the principle of embeddings. Embeddings are a further innovation in word representations, which derives from the oft-quoted Firth [1957] - *you shall know a word by the company it keeps*.

The intuition behind this is fundamentally that words with similar meanings are likelier to occur in similar contexts simultaneously. Thus, words like 'dog' and 'cat' are likely to occur in very similar contexts, bar certain differences: and assigning them arbitrary relative positions in an n-dimensional vector space,

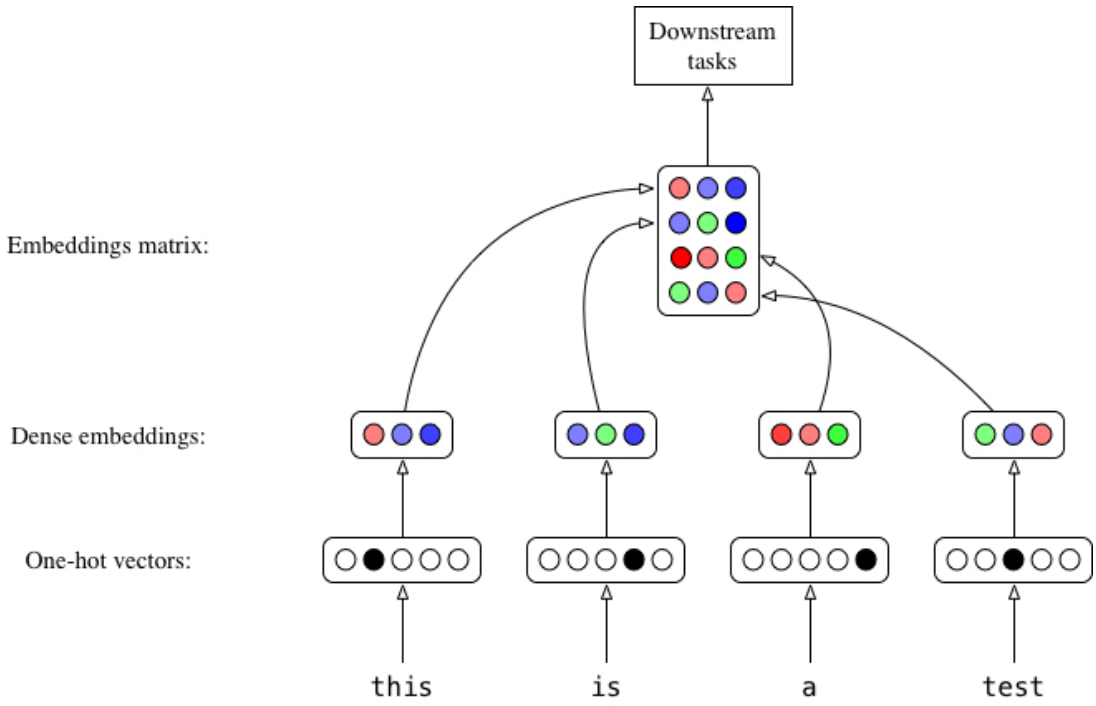

Figure 2.3: Generation of a dense (size 3) embeddings matrix from a four-word sentence, with a vocabulary size of 5.

along with other words, would mean that this meaning similarity or difference is adequately captured.

More formally, an embedding is a transformation from sparse word index vectors to dense vector word representations. Thus, initially, words are represented using *one-hot vectors* - these are fundamentally sparse vectors, that are the size of the complete relevant vocabulary or some subset of it. The specific word being considered then has its index set to a one in this sparse vector. A fully-connected layer then transforms these one-hot representations into the most appropriate dense representations for the downstream task. It is important to mention, at this point, how these embeddings are trained (which, fundamentally, means how the mappings are learnt). There are two broad approaches to training embeddings.

2.2.1 Task-specific embeddings

Depending on the task, it is obvious that embeddings can capture different sorts of distributional information, geared specifically optimising performance on the task they are trained on. Thus, intuitively, it makes more sense that they be trained specifically for a certain task, which is trivial to implement in a deep neural architecture; Figure 2.3 describes the architecture of this sort of learning system.

A significant disadvantage with using task-specific embeddings, however, is that a large number of tasks simply do not contain enough data to have ideal word representations: intuitively, mapping from one-hot vectors to dense representations ought to require large amounts of data, which most tasks cannot

provide. This often leads to more inconsistencies and poorer learning than desirable in the embeddings. This has led to the widespread use of pretrained embeddings.

2.2.2 Pretrained embeddings

Pretrained embeddings were popularised by the creation of the word2vec algorithm [Mikolov et al., 2013]. Fundamentally, pretrained embeddings consist of embeddings that have been trained independent of the task they are being used in: typically, they are trained on language modelling tasks, although embeddings trained on eg. dependency-parsed corpora do exist [Levy and Goldberg, 2014]. This is advantageous as language modelling tasks do not need any specific sort of annotation - just a large corpus is sufficient, and Wikipedia is often a fairly usable corpus. Training for language modelling is typically done one of two ways:

Continuous bag-of-words

The continuous bag-of-words (CBOW) model consists of trying to output probabilities for a target word, given a context window. For instance, given a sentence - say *the quick brown fox jumps over the lazy dog* - and a window size of 2 words to either side of the target - the algorithm attempts to predict the probability of, for instance, *fox*, given *quick brown* and *jumps over* as the relevant context. Backpropagation then attempts to fit these probabilities to resemble empirical ones sampled from the corpus as well as possible: thus, intuitively, our network learns the appropriate weights for every predicted word, and words in similar contexts would result in similar weights being used. Our weights layer is, therefore, our 'embeddings' matrix.

Skip-gram

The skip-gram embedding training system is the opposite of CBOW - it attempts to model the probabilities of certain words occuring in the *context* of a given word. The advantage of the skip-gram model is that it works better for infrequent words: there is no averaging as in the CBOW model. However, the lack of averaging also means that it is considerably slower to train. Figure 2.4 shows the architecture of both the skip-gram and the CBOW model.

2.3 Recurrent neural networks

2.3.1 Naive RNNs

The evolution of neural network architecture to support extremely deep architectures has led to the widespread use, particularly in NLP and related domains, of *recurrent* neural networks. These are, as is fairly evident from the name, recurrent structures: a recurrent layer consists of an element called a cell, as shown in Figure 2.5. Each cell feeds its own output to the next cell, which combines the output from the previous cell (hence the 'recurrent') and an independent input, into an output: this output is both accessible as an independent value, and is fed as one of the inputs to the next cell. Formally, if the output state (called the

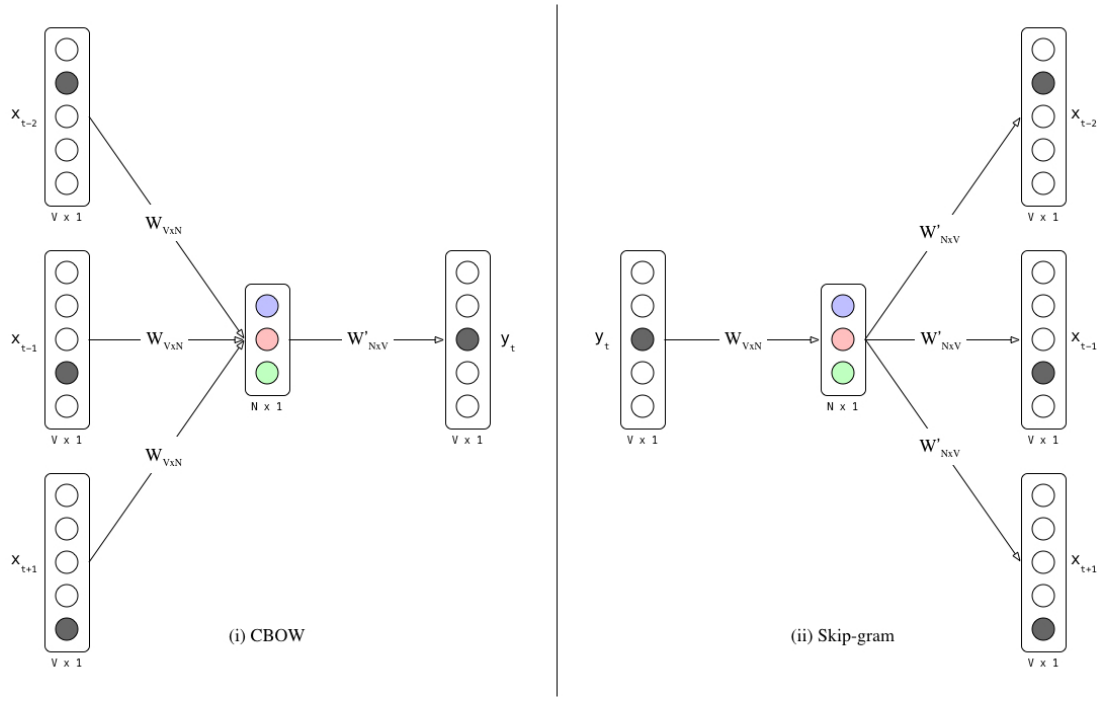

Figure 2.4: CBOW vs. skip-gram models for a vocabulary of size 5, context of size 3 and embedding dimension of size 3

hidden state) of a cell at time step t is given by $h^{(t)}$, and the input at time step t by $x^{(t)}$, we have:

$$h^{(t)} \leftarrow f(h^{(t-1)}, x^{(t)}; \theta)$$

where f is an activation function, the neuron for which is parameterised by some θ.

An RNN is thus, unlike regular deep layers, capable of retaining some sense of context: it is immediately clear why RNNs are useful in NLP, as context is often very important to represent constructions made of a sequence of elements, such as words (composed of letters) or sentences (composed of words). RNNs do not completely neglect history, but instead pass it along: the composition of this passed-along history with the current input results in representations that are highly advantageous to many tasks.

The architecture of an RNN layer ought to make it instantly obvious that their left-to-right nature is far from inherent: indeed, it is possible to flip recursive layers to pass context backwards, or, far more commonly, to combine both directions to obtain bidirectional element representations.

There is, however, a significant flaw with naive RNNs that improved architectures have attempted to alleviate.

2.3.2 Vanishing gradients, and onward

The famous *vanishing gradient* problem [Pascanu et al., 2013, Hochreiter, 1998] - or, indeed, its counterpart, the *exploding* gradient problem - is a significant stumbling block that recurrent neural networks face. Put succinctly, the issue is

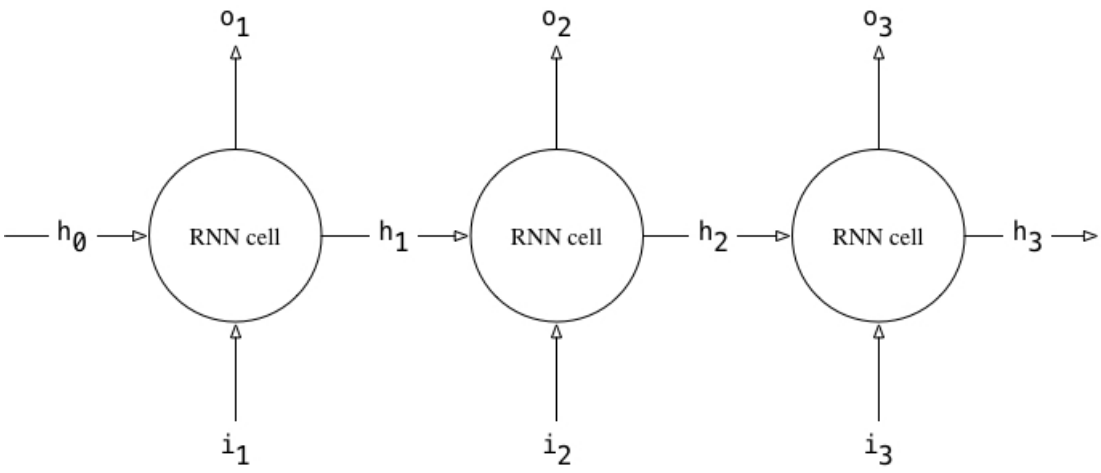

Figure 2.5: Simple RNN block with three RNN cells; h_0 is the initial hidden state which can either be learnt or initialised randomly

this: most commonly-used activation functions, such as the hyperbolic tangent ($tanh$) activation, have derivatives in the range $(0, 1)$. Given that gradients for backpropagation are calculated by means of the chain rule, it becomes fairly obvious that a series of such activation functions would, very often, result in rapidly shrinking gradients, as numbers ≤ 1 are continuously multiplied with each other. This results in the cells that appear later along the recurrent chain to train relatively extremely quickly, whilst the cells along the back train extremely slowly, sometimes not at all.

Clearly, the fix for this is to retain some sort of unchanging 'memory' of the previous state, i.e. an element with a derivative of 1; this is known as a 'constant error carousel'. The name is fairly self-explanatory: picture a carousel that carries information forward from the previous cell, with relatively stable derivatives that do not decompose to massive chains of products. Recurrent layers with such a carousel possess, in addition to a hidden state, a 'cell state' - which is a state that carries *relatively* unchanged information down the recurrent chain. Interactions with the cell state are linear, which allows the cell state to learn without suffering from vanishing gradients.

2.3.3 Long short-term memory

Long short-term memory networks, or LSTMs, were first described by [Hochreiter and Schmidhuber, 1997], in an attempt to solve the vanishing gradient problem (which they do admirably).

Whilst seemingly significantly more arcane than naive RNNs, LSTMs are fairly simple, conceptually. As is visible from Figure 2.6, there are several non-linearities (that we refer to, here, as 'gates') that are applied within an LSTM cell. These are fairly intuitively named: the input gate (i), the output gate (o) and the forget gate (f). Each of these gates initially calculates a combination and non-linear transform over the previous hidden state and the current input, similar to naive recurrent networks: $f(h^{(t-1)}, x^{(t)})$, which expands (assuming both weights

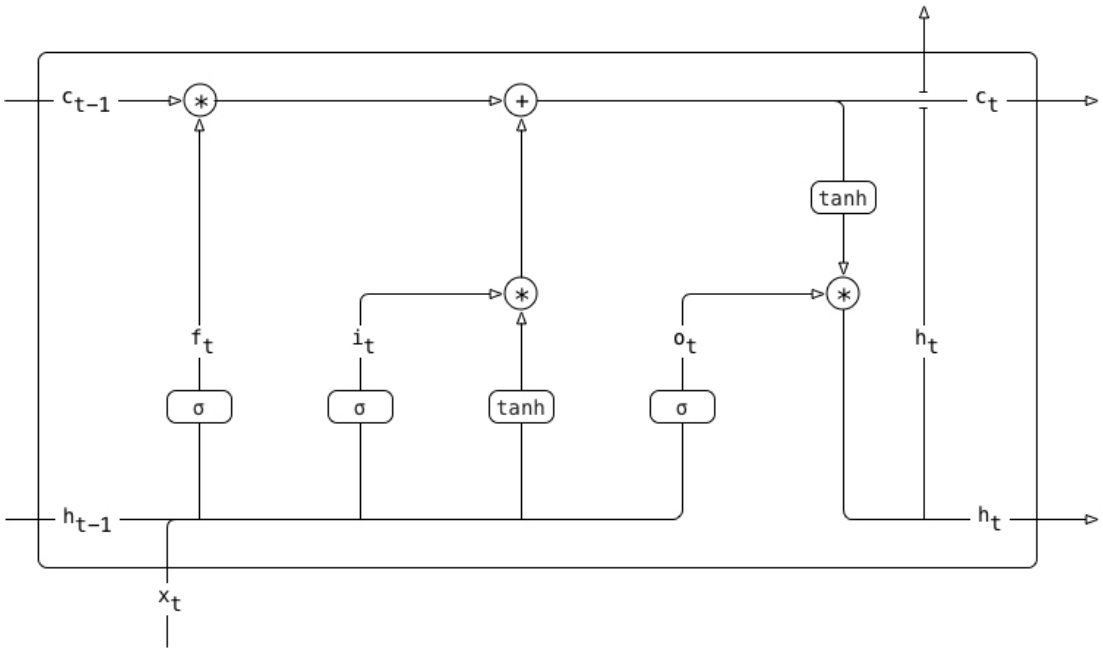

Figure 2.6: Architecture of an LSTM cell. Flow is indicated with arrows; lines are labelled appropriately with intersecting text

and biases exist) to $g(Wh^{(t-1)} + Vx^{(t)} + b)$, where g is a non-linear activation function. The weights and biases used at every gate are, obviously, different (this is indicated in the formal maths).

Typically, the activation function used for each gate is the sigmoid function σ: the reason is made clear below.

The way every gate functions is, fundamentally, to output a value between 0 and 1[2] - the range of the sigmoid function - to 'control' the amount of a certain variable. Consider, initially, the forget gate. Its output, post the application of a sigmoid function with range $(0, 1)$, is multiplied with the cell state from the previous cell: this scales the cell state, and defines how much information from the previous cell is to be retained.[3] Outputs closer to 1 in the forget vector indicate that that particular element is worth remembering, whilst outputs closer to 0 indicate that they ought to be forgotten.

Next, the input gate applies a similar sigmoid, whilst also simultaneously applying another independent *tanh*. The input sigmoid is then multiplied by the input *tanh*. This is also a fairly intuitive step: the sigmoided inputs learn how much of the input ought to be remembered, whilst the inputs that pass through *tanh* define *what* is to be remembered. *tanh* is used largely because of the stability of its derivative. The product of the two is then added to the cell state.

Finally, the cell state passes through a *tanh* - merely to introduce an essential non-linearity - and is then multiplied by the sigmoided value of the output gate. This output is then considered the hidden state of that particular LSTM cell. The *tanh*, therefore, serves purely to transform the cell state into the hidden

[2]Note that, whilst not explicitly mentioned, these gates do learn what to output to optimise performance

[3]Neither the forget gate's output, nor the previous cell state, are integers: they are vectors

state via a non-linearity, whilst the sigmoid serves to indicate what parts of that transformed cell state are truly relevant and ought to be captured in the hidden state.

This can formally be expressed by a set of equations; at a particular time step t:

$$
\begin{aligned}
i^{(t)} &\leftarrow \sigma(W_i x^{(t)} + V_i x^{(t)} + b_i) \\
o^{(t)} &\leftarrow \sigma(W_o x^{(t)} + V_o x^{(t)} + b_o) \\
f^{(t)} &\leftarrow \sigma(W_f x^{(t)} + V_f x^{(t)} + b_f) \\
c^{(t)} &\leftarrow y^{(t)} \cdot c^{(t-1)} + i^{(t)} \cdot \tanh(W_y x^{(t)} + V_y x^{(t)} + b_y) \\
h^{(t)} &\leftarrow o^{(t)} \cdot \tanh(c^{(t)})
\end{aligned}
$$

Note that, in these equations, the suffix i indicates parameters relevant to the input gate, o to the output gate, f to the forget gate and y to the input gate's *tanh* transform.

2.3.4 Character-level RNNs

Character-level RNNs are an improvement on character-level language models, à la Kim et al. [2016]. Fundamentally, a character-level language model attempts to augment an existing word-level model with character-level 'information'; i.e. by representing words as compositions of vectors that represent the constituent characters in the word. The biggest advantage of this is instantly visible for morphologically complex languages: augmenting these systems with character-level information allows for some semblance of similarity in representations of words that are distinct in word space, if they are grammatically similar: similar affixes would lead to their vectors being pushed closer to each other.

Where character-level models really shine, however, is where the word embedding vocabulary is restricted, and smaller than the vocabulary of the actual test data. In these situations, rather than using an (often randomly initialised) 'unknown' vector, utilising character level information helps provide *some* information, based, often, solely on affixes, that unknown vectors certainly do not provide. Whilst most character-level models in the literature use some variety of convolutional network [Chiu and Nichols, 2015, Zhang et al., 2015, Vosoughi et al., 2016], we instead rely on multiple levels of a recurrent neural network, à la Dozat et al. [2017].

Fundamentally, it is important to remember that there needs to be some element in our architecture that compresses the output *per character* into a single vector; whilst convolutional architectures work well for this sort of compression, so do recurrent architectures. We therefore pass our character embeddings - which are calculated similar to word embeddings, but with other characteres as context - into an LSTM, the last state of which is concatenated to self-attention over the rest of the hidden states[4] passed to a multi-layer perceptron. The vector that this MLP outputs is what we consider representative of the 'composition' of the characters in our word or token. This is exemplified in Figure 2.7.

[4]An explanation of various forms of attention is beyond the scope of this thesis; we refer the reader to Bahdanau et al. [2014] and Lin et al. [2017] for further reading.

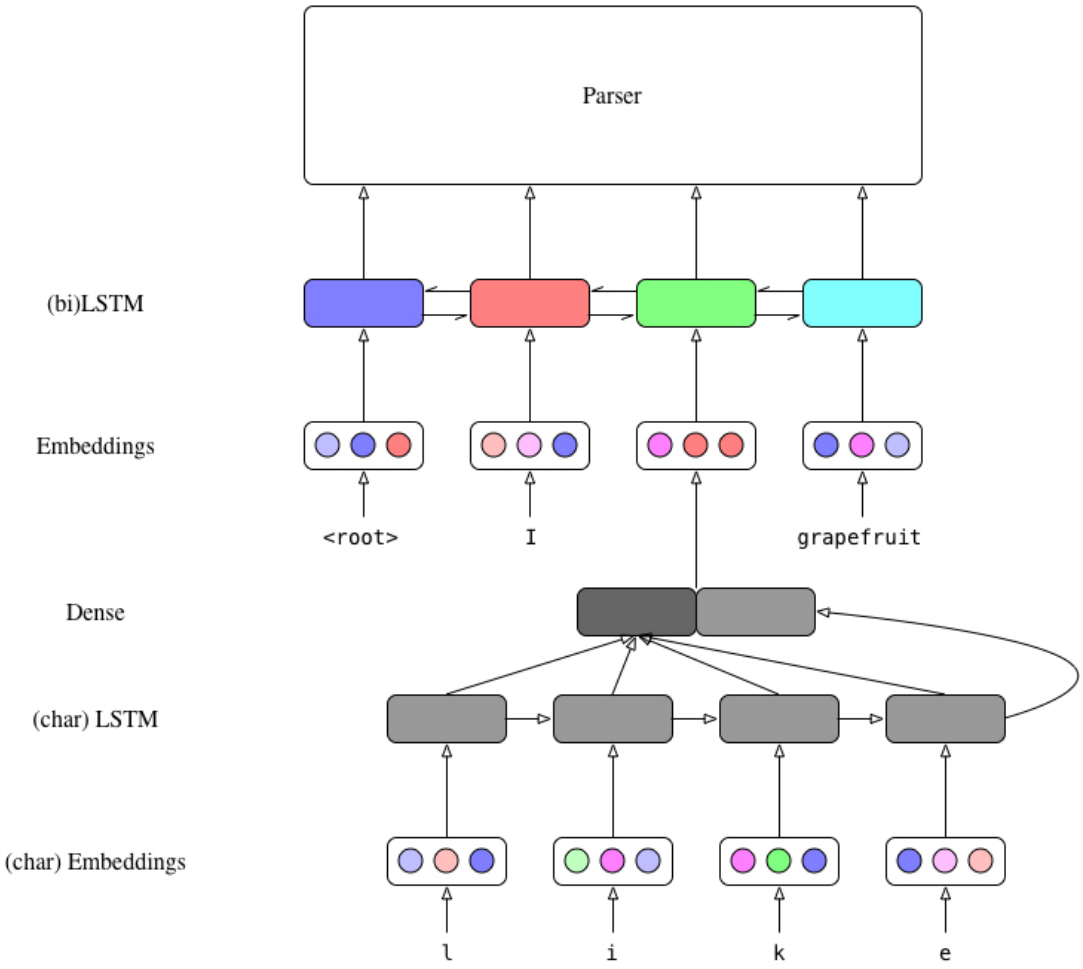

Figure 2.7: A composition of characters for the word 'like' being fed upstream to our parser (which is described better in Figure 2.10)

2.4 Multi-task learning

A fairly significant part of this thesis relies on the principle of multi-task learning. The recent Ruder [2017] is an excellent overview of the domain and the state-of-the-art in the field; however, we summarise key learnings from MTL briefly here.

Fundamentally, multi-task learning relies on the intuition that humans use information and knowledge from tasks they already know to accomplish other tasks. This can transfer to deep learning: by sharing layers in a network across a variety of tasks, the shared layers attempt to adapt to each task. This adaptation winds up providing the shared layers with information that, in turn, winds up heavily influencing performance on the first task.

In its most basic setting (Figure 2.8), typically referred to as 'hard parameter sharing', layers are shared across tasks: each shared layer learns representations that benefit both tasks, whilst task-specific layers allow these representations to specialise to that particular task.

Soft parameter sharing (Figure 2.9), on the other hand, has independent networks for each task: the parameters that are meant to be shared are then regu-

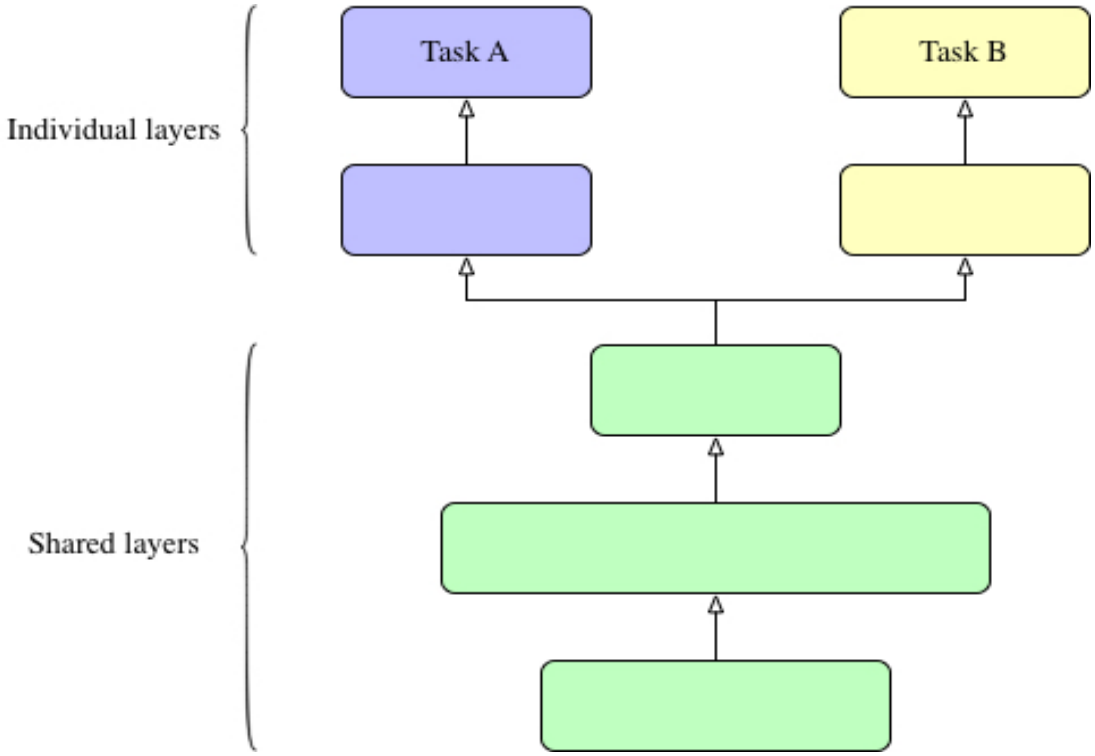

Figure 2.8: Hard sharing parameters for two tasks

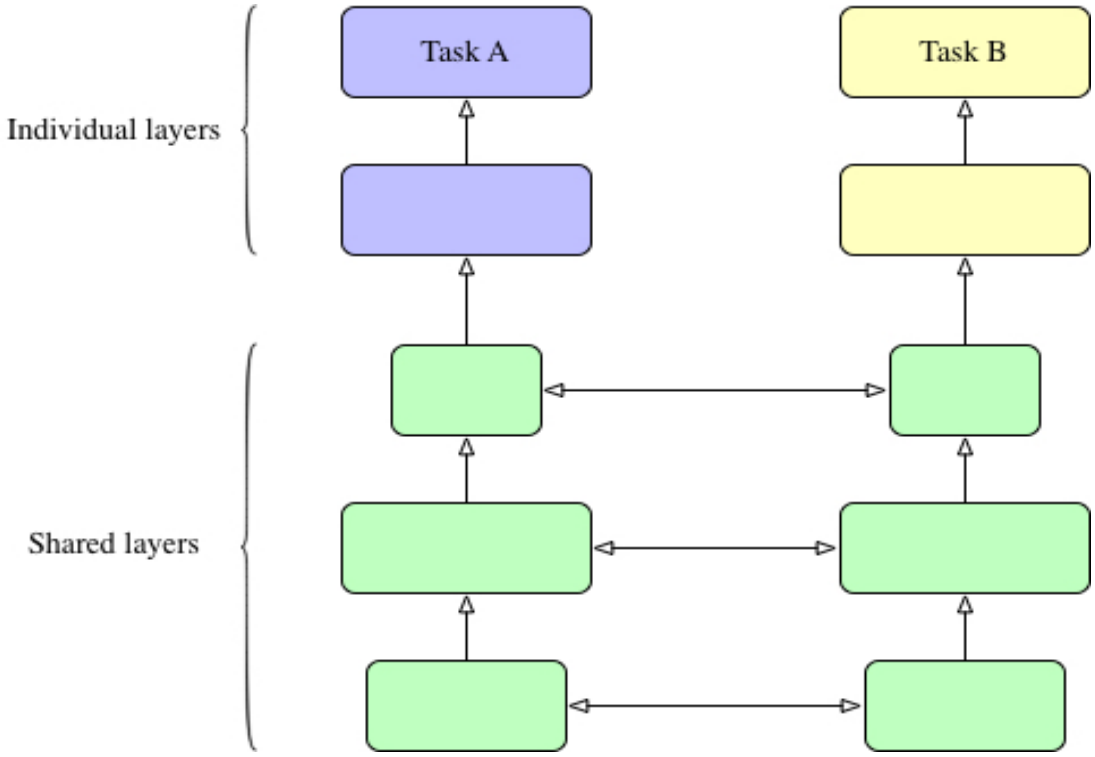

Figure 2.9: Soft parameter sharing for the same tasks

larised using some form of distance metric.

There are other proposed architectures for multi-task learning, many of which improve on the hard/soft-parameter sharing paradigm. Worthy of note are, in particular, cross-stitch networks [Misra et al., 2016] and sluice networks [Ruder et al., 2017a]; we do not, however, implement either of these networks due to their effectivity being more visible with tasks that involve sharing more layers than we do.

Bingel and Søgaard [2017] provide a review of specifically what tasks help, and attempt to quantify how much they help, when jointly learnt in a multi-task environment. We attempt to extend on this work by evaluating certain tasks that are possible only within the domain of code-switching, and attempt to quantify their usefulness.

2.5 Dependency parser

2.5.1 Architecture

Our dependency parser is, more or less, a reimplementation of the graph-based parser created by Dozat and Manning [2016]. Reimplementing the parser, whilst seemingly unnecessarily tedious, allows us to modify the architecture to take advantage of a variety of our code-switching specific settings.

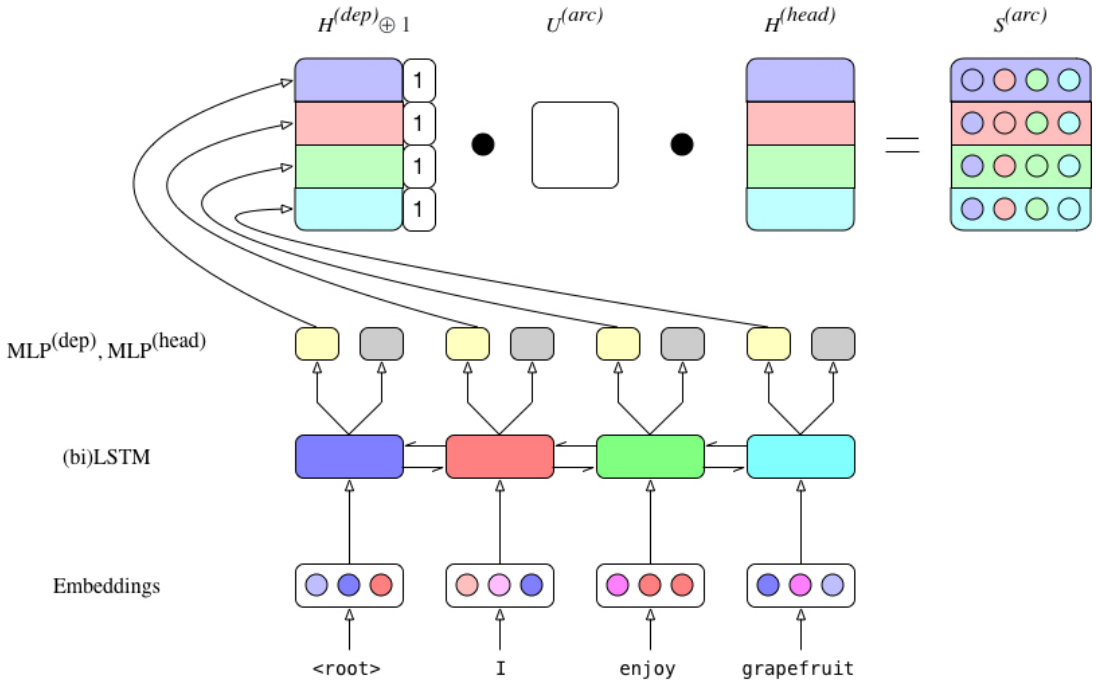

Figure 2.10: Architecture of our dependency parser; colours indicate specific tokens

Figure 2.10 is the broad architecture of the dependency parser. Whilst seemingly complex, the way it functions is fairly intuitive.

Initially, our input tokens - both word forms and POS tags - are passed through an embeddings layer, that converts their one-hot representation to dense

embeddings. These embeddings are then (as is the norm for a large number of NLP tasks) passed through a bidirectional LSTM, which generates vectors that take context into account. Note that this is a multilayer LSTM: fundamentally, this works the same as a deep neural network, with the 'deepness' being both a horizontal phenomenon (multiple layers for multiple time-steps) and a vertical one (multiple LSTMs stacked on top of each other).

The hidden state of every LSTM cell in the final LSTM layer is then passed to two separate MLP[5]. For each cell (each of which is meant to represent a token), one MLP is meant to 'learn' representations of that token as the head of a dependency relation between two words, whilst the other is meant to learn representations of the token as a dependent.

Next, a bilinear transformation is then applied to the stacked outputs of each series of MLP. A bilinear transformation is, in principle, similar to a linear one, but relies on transforming multiple matrices with a single weight and bias. Put more formally, the bilinear analog to the linear transformation $Wx+b$ is x_1Wx_2+b.

The bilinear transformation fundamentally results in a square matrix, with each element representing the probability of a token (represented by the row) being the head of the token (represented by the column). This is demonstrated in Figure 2.10 using colours: each colour represents a particular word. A softmax layer is then applied to obtain numerically stable probabilities for arcs for every possible $(head, word)$ pair.

Having obtained these arc probabilities, we then attempt to obtain deprel probabilities. The network is fundamentally the same until after the MLP stage (note that two different MLPs are used here, as the representations learnt for labels are not necessarily the same as those learnt for heads). The bilinear step is rather different: we do not need to combine every possible token (as head) with every possible token (as dependent) as we already have the most likely heads. We therefore populate the deprel vector after initially selecting the appropriate most-probable head, and all dependents. We do not select the 'most probable' dependent as a single head can have multiple dependendents, and we calculate the likeliest relation with all such dependents, eventually selecting the one that represents the appropriate dependent.

Formally, our arc predictor can be represented by the equations:

$$\mathbf{s_i}^{(arc)} = H^{(head)}\, W^{(arc)} \mathbf{h}_i^{(dep)} + H^{(head)} \mathbf{b}^{\top(arc)}$$
$$y_i'^{(arc)} = \arg\max_j s_{ij}^{(arc)}$$

The biaffine transformation used by the deprel predictor is more complex, and is meant to use multiple biases:

[5]MLPs are fundamentally multiple layers of densely connected neurons

$$
\begin{aligned}
\mathbf{s}_i^{(rel)} &= \mathbf{h}_{y_i'^{(arc)}}^{\top(head)} \mathbf{U}^{(rel)} \mathbf{h}_i^{(dep)} \\
&\quad + W^{(rel)}(\mathbf{h}_i^{(dep)} \oplus \mathbf{h}_{y_i'^{(arc)}}^{\top(head)}) \\
&\quad + \mathbf{b}^{(rel)} \\
y_i'^{(arc)} &= \arg\max_j s_{ij}^{(arc)}
\end{aligned}
$$

Having obtained matrices indicating the softmax probabilities of every possible arc (and the labels associated with each 'best arc', we draw the reader's attention to the fact that this sort of unstructured, element-by-element prediction inherently fails to guarantee that any output would obey the tree constraint; we are, fundamentally, predicting graphs (hence the name). We therefore attempt to find a spanning tree through the graph: specifically, the maximum spanning tree (as softmax probabilities are proportional to actual probabilities). To do so, we employ the widely-used Chu-Liu-Edmonds algorithm [Chu and Liu, 1965a, Edmonds, 1967], which recursively returns a spanning tree for a weighted graph; our weights are our probabilities, and the specially designed root node is the head of the tree.[6]

For our biaffine transforms, we referred to `https://github.com/chantera/biaffineparser` for pragmatic implementation help.

2.5.2 Implementation

We re-implemented the parser in PyTorch [Paszke et al., 2017], specifically version 0.3.1. Our main motivation in reimplementing it was to make it easier to further augment the network with our own architectures. The source code for our implementation is freely available for review on GitHub, at `https://github.com/vin-ivar/vinparser`. We use several libraries to help make dealing with CoNLL files easier, primarily the excellent `torchtext` library[7], built specifically for vectorising raw text; as it deals only with specific file formats, we wrote several converters to convert our CoNLL-U files to the relevant CSVs. To ensure replicability of our initialised hyperparameters, we fix all internal random number generation seeds to the numeric value 1337.

[6] We use CLE code from the original parser implementation: `https://github.com/tdozat/Parser-v1`

[7] `https://github.com/pytorch/text`

2.5.3 Hyperparameters and optimisation

Parameter	Value
Embedding dim.	100
LSTM hidden state dim.*	500
LSTM layers	3
Arc MLP dim.	400
Label MLP dim.*	200
Dropout	33%
Batch size*	40
Epochs*	10
Learning rate*	$1 * 10^{-3}$
Char. embedding dim.	100
Char. LSTM dim.*	150
LSTM dropout	33%
Attention dropout*	0%

Table 2.1: Hyperparameters for our parser; ones that differ from the original are indicated with asterisks

We retained most of the hyperparameters used in the original Dozat and Manning [2016] parser, albeit with certain modifications that we optimised towards using grid search. Our hyperparameters for an additional character-based LSTM layer [Dozat et al., 2017] were also similar. Whilst the original hyperparameters were optimised for monolingual parsing on English, we used them only as a starting point, then added several tweaks to measure downstream performance on our datasets. Other differences primarily involve training: whilst the original parser uses minibatches determined by token counts, we use sentence counts. Further, we fix our training period to ten epochs, and do not implement early stopping or annealing. Our hyperparameters are described in Table 2.1. Our character hyperparameters were more optimal with significantly smaller recurrent hidden state dimension sizes: 150 instead of 300. We hypothesise that this is due to the fact that Devanagari for Hindi has more rigid grapheme-to-morpheme mappings than Latin for English.

3. Parsing for Code-Switched Languages

In this chapter, we outline our approaches to modifying our dependency parser to work with code-switched languages. As such, this chapter is broadly divided into several 'sub'-chapters.

i. Data and baselines

In this section, we describe the validation and test data that we use, along with several baselines that we define: these baselines are fairly intuitive, but despite this, as we shall observe in a summary of our results, they perform exceptionally well.

ii. Word representations

In this section, we describe our experiments with a variety of word representations, mapped to induce multilinguality. We compare several approaches to inducing these mappings, and compare and analyse the results.

iii. Treebank-level modifications

Here, we describe a set of partially deterministic alterations that we apply to our treebanks, in an attempt to simulate code-switching where it does not exist.

iv. Network alterations

By far the most significant contribution of this thesis, this section describes techniques we use to alter our network to better cope with code-switched language.

3.1 Data and baselines

In this section, we describe the treebanks we are using, and provide some detailed statistics on distributions of syntactic annotation in these treebanks.

3.1.1 Treebanks

Whilst we have referenced the data we intend to use in Section 1.4.2, we describe this data in more detail here. Fundamentally, we evaluate all our parsing experiments on two datasets: a Hindi/English code-switched dataset, and a Komi/Russian one. Once again, we wish to emphasise the main aim of our thesis, which is **relying on monolingual data for training**. As such, our data fundamentally decomposes into two 'collections':

1. Monolingual training data for both the languages that constitute the code-switched pair

2. Code-switched development/test data

Further, several of the approaches that we describe involve augmenting monolingual training data with some development data; it thus becomes important for us to have separate dev/test splits.

For our Hindi/English suite, we use the data described in Bhat et al. [2017]; for Komi/Russian, we use the data described in Partanen et al. [2018]. Whilst our Hindi/English data comes pre-divided into dev/test sets, we lack development data for Komi. Indeed, our Komi test data is divided into three sections: a monolingual Komi corpus, a code-switched corpus created by transcribing spoken Komi, and another code-switched corpus created by manually modifying a monolingual Komi corpus. We ignore the first of these three splits for all our experiments. Further, we append the rest of these splits to create a single test corpus. Finally, we do *not* use development data to fix hyperparameters for Komi: we fix our hyperparameters on our Hindi/English development data and use the same for Russian/Komi. For tasks where we require a development fold (described later), we use the spoken Komi corpus as our dev set: this is primarily motivated by the fact that it is smaller, and a larger test set gives us a more confident evaluation.

Note that our Hindi/English data comes pre-transliterated; as most of the data in the code-switched corpus was scraped from Twitter or Facebook, the original sentences follow a non-standardised transliteration scheme *(Romanagari)*.

There are several analytical statistical metrics that apply to both monolingual and cross-switched treebanks; most of these are rather intuitive, and are provided without explanation in Table 3.1. Note that these stats do not necessarily match those supplied on the Universal Dependencies website, as we only use the training split of our monolingual treebanks.

3.1.2 Code-switching statistics

In this section, we draw the attention of the reader to Guzmán et al. [2017], who introduce several excellent analytical metrics that attempt to describe code-switched corpora. In this section, we briefly describe these metrics; we leave out

Metric	en-hi			kpv-ru		
	English	Hindi	Bilingual	Russian	Komi	Bilingual
Sentences	12K	13K	448	3.8K	40	105
Tokens	204K	281K	6.7K	76K	363	893

Table 3.1: Basic analytical statistics for our treebanks (combined dev and test splits

mathematical formalisms for brevity, and urge the reader to refer to the original instead.

Language normalisation

Prior to calculating our metrics, we introduce modified variants of our corpora, by introducing what we call *language normalisation*; this, fundamentally, involves us replacing all our 'extra' language labels – such as labels that mark punctuation, or foreign words, or linguistically ambiguous words – with the previous 'valid' language tag, if it exists, or the next, if not (such as when these tokens are sentence-initial).

Metrics

Note that as the sentences in our treebanks are not necessarily contiguous, we cannot consider the entire treebank a fixed corpus, and instead average our metrics over a sentence level, where relevant. Briefly, therefore, our metrics are:

1. **M-index**: the multilingual (or M-) index 'quantifies the inequality of the distribution of language tags in a corpus of at least two languages'. Intuitively, it is bounded by 0, indicating a monolingual corpus, and 1, indicating that every language in the corpus has the same number of tokens in the corpus.

2. **Language entropy**: this metric relies on the well-known concept of Shannon entropy [Shannon, 2001], and extends it to calculate the entropy over the languages in the corpus, i.e. the number of bits needed to represent the language distribution. It is bounded by 0 and $\log_2(k)$, where k is the number of languages in the corpus.

3. **I-index**: the probability of switching; the I-index is a fairly simple metric that averages the number of switches over the number of tokens.

4. **Burstiness**: one of the more complex metrics we use, *burstiness* fundamentally describes the divergence of code-switching behaviour in a corpus from a Poisson process, i.e. from completely random code-switching. A value closer to the lower bound of -1 indicates periodic code-switching behaviour, whilst values closer to the upper bound of 1 indicate randomness in switching.

5. **Span entropy**: a modification of language entropy, span entropy instead describes how many bits are necessary to describe the distribution over language *spans*, i.e. contiguous monolingual chunk lengths.

Cross-linguistic arc fertility

In this section, we introduce a concept we name 'cross-linguistic arc fertility'. Put quite simply, this refers solely to the existence of deprels across languages. It is very apparent, intuitively, that given the monolingual nature of every *sentence* in our baseline[1], that our parser should never see arcs across tokens that belong to different languages. This is, obviously, an issue: real-world code-switched data, when appropriately annotated, is bound to have cross-linguistic arcs. Indeed, as we demonstrate below, cross-linguistic arcs are remarkably frequent, in the vicinity of almost 50% of all our arcs.

It quickly becomes clear, from these figures, that one of our key focuses ought to be improving our parser's ability to recall arcs across languages, or, to use our newly coined term - to demonstrate greater cross-linguistic arc fertility.

They are presented for both our standard corpora and our language-normalised corpora in Table 3.1.2. We do not normalise our Komi/Russian data as there are no extra language labels; indeed, we have only one language marker in our data, for Russian tokens: the assumption is that every unmarked token is in Komi.

	en-hi				**kpv-ru**
	Orig. dev	Norm. dev	Orig. test	Norm. test	Test
M-index	0.35	0.97	0.44	0.98	0.42
Language entropy	1.68	0.99	1.65	0.99	1.03
I-index	0.43	0.24	0.43	0.25	0.36
Burstiness	-0.03	0.10	-0.03	0.11	-0.21
Span entropy	1.60	1.73	1.38	1.45	1.14
Cross-linguistic arc fertility	0.61	0.45	0.61	0.50	0.42

Table 3.2: Code-switching statistics for our treebanks; note that Komi/Russian have only a single unified, normalised test treebank

These metrics, interestingly, seem to indicate that our Russian/Komi corpora function very similarly to our Hindi/English ones wrt code-switching, although they also seem to indicate that code-switching in Komi appears to be more predictable. These metrics also, pleasingly, appear to be similar to the metrics for the *Killer Crónicas* corpus mentioned in the source paper [Guzmán et al., 2017], which is a corpus of emails written in 'Spanglish' (Spanish/English).

3.1.3 Constructing a baseline

Our baseline (which, as we demonstrate, is surprisingly intelligent), consists of what is fundamentally a naive concatenation of the *monolingual* treebanks corresponding to the code-switched data in our corpus. For Hindi/English, therefore, these are the monolingual corpora, unexcitingly named UD_English and UD_Hindi. For Komi and Russian, we use the standard UD_Russian corpus; for Komi, we use the 40 sentence training dataset prepared by Partanen et al. [2018].

We then proceed to train our baseline parser on this concatenation; we further compare the performance of our parser both with and without character-level features included. Our results are outlined in Table 3.3.

[1]Our baseline training data is multilingual, but never at a sentence level

		UAS	LAS	wLAS
en-hi		**76.05**	**60.15**	**52.27**
en-hi	[+char]	73.57	56.84	48.55
ru-kpv		64.95	52.86	49.68
ru-kpv	[+char]	**65.40**	**53.08**	**49.97**

Table 3.3: Baselines, with and without character representations included

It is interesting to note, here, that our performance on Komi/Russian improves with the addition of character features, whilst the performance on Hindi/English heavily deteriorates (and this, indeed, is a recurring theme through our experiments). We hypothesise that this is due to the difference in scripts in English and Hindi; our parser is unable to learn any useful morphological information it can leverage. The use of the Cyrillic script in both Russian and Komi, along with use of Komi affixes with Russian tokens during code-switching, make character features significantly more helpful: not only do they learn to observe patterns across both languages, they also learn to use morphological information that occurs monolingually.

3.2 Word representations

In this section, we describe our experiments with word representations, more specifically with mapped word embeddings. We begin with an explanation of precisely what mapped embeddings are, along with some background literature, in Section 3.2.1. We then explain our motivation for using mapped embeddings by referring to another experiment involving monolingual dependency parsing using mapped embeddings[2]. Finally, we describe how we apply these mapped embeddings to our own work, and our results.

3.2.1 Mapped embeddings

The concept of mapping embedding spaces fundamentally involves applying transformations to a given embedding space E, to shift it to 'align' with another distinct space F. This is a practical implementation of the fundamental intuition that embedding spaces are *isomorphic*: i.e., embedding spaces even for different languages 'look' similar, based on the intuition that all languages convey the same thematic information. Ruder et al. [2017b] provide a solid review of most cross-linguistic embedding projection models that existed at the time of the paper being written; however, there have been newer techniques that deal with even more resource-scarce situations that have been proposed since.

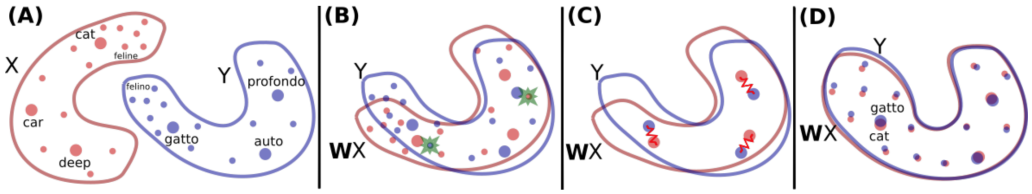

Figure 3.1: Conneau et al.'s [2017b] approach to adversarially learning embedding mappings

The main techniques for mapping that we draw the attention of the reader to are the two 'seminal' works in the field - Conneau et al. [2017b] and Artetxe et al. [2016]. The former propose using an adversarial learning system, that involves one system attempting to 'guess' what language a particular embedding belongs to, with the other attempting to align the spaces to prevent the guesser from being successful. Figure 3.1 (taken from the original paper) briefly outlines how this works.

Artetxe et al. [2016] attempt to do much the same thing, but with a much simpler architecture; they attempt to learn, given word embedding matrices X and Z, representing the 'source' and 'target' spaces, a transformation matrix W, which attempts to bring X as close to Z as possible.

More formally, they seek to minimise, for every element i in our seed list:

[2]This subsection was joint work with Artur Kulmizev and Mostafa Abdou, and is to be published in the near future.

$$\sum_i ||X_i W - Z_i||^2$$

i.e., evaluate

$$\arg\max_W \sum_i ||X_i W - Z_i||^2$$

Other techniques, also useful to us, attempt to model these transformations in the absence of even the relatively small seed lists the latter of these techniques requires; specifically, Artetxe et al. [2017] and Artetxe et al. [2018], which use either easily available parallel data (like numbers), or no parallel data at all. We compare both of these on our Komi/Russian treebank.

Recent research points out the limitations of Conneau et al.'s [2017b] work, particularly relevant to underresourced languages [Søgaard et al., 2018]; we therefore restrict our analyses to the other mapping techniques described in this section.

3.2.2 Cross-lingual dependency parsing

In this section, we introduce a novel experiment on *cross-lingual dependency parsing*. Specifically, this task attempts to answer the question: how far can we get with parsing a treebank, given treebanks for similar languages, and embedding mappings? In order to evaluate this, we pick four treebanks for a UD language family (say Romance languages). Amongst these languages, we select one (say Catalan) to be our target 'underresourced' language. We then map word representations from our other (source) languages of the same family (in this specific example: Spanish, Portuguese and French) to the same space as the target language. Finally, we attempt to see how far training on each individual source language (or a combination of all three) gets us, with parsing our target language. In doing so, we also gradually vary the size of target training data that we augment our cross-lingual experiments with. This is similar in spirit to Ammar et al. [2016]'s work on parsing many languages based on leveraging linguistic data, albeit with mapped embeddings replacing many of their features.

Our results for this experiment are quite positive, and outlined in Table 3.4. Table 3.5 describes the baselines for parsing Catalan with the same fixed amount of data, but no data from any other language.

Our motivation in evaluating cross-lingual embeddings on code-switched data quickly becomes clear. Indeed, this is similar in spirit to experiments carried out in Partanen et al. [2018], albeit evaluated on multiple mapping systems. Fundamentally, mapping embedding spaces ought to increase our cross-linguistic arc-fertility: to understand this, it is crucial to note that *our parser never actually sees a 'word'* – all it sees is an *embedding* representing the word. Mapping our embedding spaces to overlap, therefore, means that our parser has absolutely no way of knowing what language a word comes from, but only its distributional representation – which, further, the parser has no way of knowing what language it belongs to. Assuming (and, indeed, relying on the fact) that our embedding graphs for both languages are isomorphic, we expect our parser to treat code-switched data as a valid vector-space combination of both languages.

		Catalan proportion				
		0	1/16	1/8	1/2	1
Spanish	UAS	0.57	0.83	0.86	0.90	0.92
	LAS	0.36	0.76	0.80	0.86	0.88
French	UAS	0.30	0.82	0.86	0.90	0.92
	LAS	0.12	0.73	0.79	0.86	0.88
Portuguese	UAS	0.29	0.82	0.86	0.90	0.92
	LAS	0.10	0.74	0.80	0.85	0.88
All	UAS	0.34	0.85	0.87	0.90	0.91
	LAS	0.19	0.77	0.80	0.85	0.87

Table 3.4: Results for parsing Catalan with various levels of training data augmentation

	Catalan proportion				
	0	1/16	1/8	1/2	1
UAS	-	0.58	0.72	0.85	0.90
LAS	-	0.35	0.59	0.79	0.85

Table 3.5: Results for parsing Catalan without other mapped training data

3.2.3 Embeddings and methods

The specific embeddings that we use are size 300 fastText embeddings [Bojanowski et al., 2017], trained on Wikipedia for Komi, Russian and English; for Hindi, our embeddings are trained on a combination of Wikipedia and Common Crawl data [Grave et al., 2018]. This is largely due to the size of the Hindi Wikipedia: we obtain significantly improved performance by augmenting it with Common Crawl data. Whilst we would have preferred to do the same for Komi, we did not have easy access to Common Crawl data in Komi, and had to rely on the (admittedly poor) Wikipedia-learnt embeddings.

At this point, one discrepancy might stand out: in Section 2.5.3, we mentioned that our parser used size 100 word embeddings. This is true; we add a compression layer that we did not mention in the implementation section, that compressed our size 300 fastText vectors into vectors of length 100. These are then added and/or concatenated with other vectors as described in Section 2.5.1. We describe results for both our compressed and uncompressed embeddings here; note, however, that we use compressed embeddings for all further experiments.

We use three forms of mappings: for Hindi/English, we use the supervised embedding mapping method (that uses a seedlist) described by Artetxe et al. [2016]. The Apertium project [Forcada et al., 2011], a collection of rule-based machine translation pairs, uses XML dictionaries for lexical transfer from one language to another. We convert the parallel wordlist from the Apertium Hindi $\rightarrow$ English translation pair, and use this as a seed for our mapping; our wordlist consists of $40,305$ word pairs, well above the recommended minimums. Our motivation for not using the ground-truth dictionaries provided by Conneau et al. [2017b] is that they are exceptionally bad, for Hindi/English: a quick pass through the dictionary reveals that very few of the word pairs are accurate.

For Komi/Russian, we do not have parallel wordlists; we therefore use the mildly supervised method [Artetxe et al., 2017], that relies on identical words: these ought to, in theory, occur across languages that display code-switching with each other. We also refer to this method as 'mapped', despite the method being different to Hindi/English. Another mapping system we evaluate Komi/Russian on is a completely unsupervised one, that tries to rely on the relative internal structure of the embeddings [Artetxe et al., 2018].

To compare our results to another baseline, we use concatenated unmapped embeddings: conveniently, PyTorch allows us to load embeddings stored in plaintext rather than their binary variants, which means that the naive concatenation of two embedding files is also a valid embedding file. Our results for each of these mapping techniques are provided in Table 3.6; a similar evaluation with uncompressed embeddings is presented in Table 3.7.

en-hi		**UAS**	**LAS**	**wLAS**
Concatenated		76.93	**61.61**	**53.81**
Concatenated	[+char]	76.72	60.91	52.49
Mapped		**77.72**	61.31	53.31
Mapped	[+char]	75.75	59.48	51.33
ru-kpv		**UAS**	**LAS**	**wLAS**
Concatenated		64.28	50.28	46.40
Concatenated	[+char]	**67.97**	**53.98**	**50.99**
Mapped		66.52	52.63	48.89
Mapped	[+char]	66.97	52.63	48.72
Unsupervised		65.29	51.96	49.34
Unsupervised	[+char]	64.61	51.51	48.24

Table 3.6: (Compressed) embedding performance

Several things instantly stand out here. For one, both variants of embeddings provide significant improvements over the baseline for Hindi; however, the difference between the two sorts is not clear, with variations across UAS and LAS; indeed, the architecture with the best results for compressed embeddings is the exact opposite to that for uncompressed ones.

For Komi, however, the difference is much clearer, with baseline concatenated embeddings clearly outperforming other variants. We hypothesise that this is for two reasons: first, the Komi embeddings are not good enough (note that we do not have parallel Komi/Russian wordlists) to suffer from the further inevitable degradation in performance when mapped, and that the mapping system itself shows several flaws, similar to those pointed out in, for eg. Søgaard et al. [2018]. However, one thing is for certain: our raw concatenated embeddings are clearly *helpful*: our performance is well over the baseline, with an almost 3% increase in UAS, and a single percentage increase in LAS.

Whilst character features help with compressed embeddings, they do not help much with uncompressed ones: this is likely due to the fact that a size 300 output vector is too large to capture any meaningful character information.

en-hi		**UAS**	**LAS**	**wLAS**
Concatenated		**77.24**	60.82	52.31
Concatenated	[+char]	76.81	60.79	52.66
Mapped		77.21	**61.43**	**53.22**
Mapped	[+char]	75.78	59.64	50.84
ru-kpv		**UAS**	**LAS**	**wLAS**
Concatenated		**67.08**	**53.75**	**50.79**
Concatenated	[+char]	63.83	50.17	45.32
Mapped		66.18	**53.75**	50.20
Mapped	[+char]	61.37	48.82	44.84
Unsupervised		65.85	53.42	49.98
Unsupervised	[+char]	59.01	45.91	41.67

Table 3.7: Comparable uncompressed embedding performance

3.3 Treebank-level modifications

In this section, we describe approaches to attempt to artifically induce code-switching in our monolingual treebanks, using a set of rules we derive from statistical distributions from our development set.

Our intuition behind this section is the fact that it is significantly likely that code-switches are governed by syntax, and therefore by dependency relations; i.e., they are likelier to occur at specific dependency relations between words. We evaluated our hypothesis by searching through our development data, looking for dependency relations at specific code-switch points; our intuition bore out. Table 3.8 summarises the frequencies of dependency relation labels at which code switches occur, for both of our treebanks.

en-hi		**ru-kpv**	
Deprel	Frequency	Deprel	Frequency
punct	13.95	punct	16.81
nmod	11.99	advmod	16.60
nsubj	10.03	nmod	9.70
case	9.48	nsubj	7.47
aux	6.98	case	7.32
advmod	6.54	obl	7.18
obj	5.47	cc	6.50
compound	5.27	amod	5.89
mark	3.40	xcomp	3.99
advcl	3.31	obj	3.65

Table 3.8: Deprel frequency per treebank

Having established that switches are overwhelmingly governed by certain markers, we then attempt to sample which of these markers specifically involve closed-class words: the intuition being that randomly sampling from the opposite treebank ought to be easier with closed-class words due to the limited space.

It is important to note, however, at this point, that these relations are not necessarily bidirectionally frequent, and any replacement strategy ought to take into account the difference in edge distribution based on directionality. This means, essentially, that our edges from L_1 to L_2 are not necessarily the same as those in both directions: any algorithm ought to take this into account. Our original statistics from Table 3.8 therefore expand into the more interesting distribution found in Table 3.9.

A very interesting phenomenon is instantly visible through this table. To highlight this better, all dependency relations where at least one of the dependents is typically closed-class word are written in red. It is instantly visible that there is a very significant imbalance between the frequency of these closed-class relations across languages in a pair. Thus, English tokens in Hindi/English treebanks are overwhelmingly likelier to be governed by closed-class Hindi tokens than the inverse; the opposite is true for Russian tokens being governed by Komi.

en-hi				kpv-ru			
→		←		→		←	
case	14.18	nmod	17.04	punct	28.90	advmod	25.05
punct	11.39	nsubj	14.08	advmod	14.28	case	13.79
aux	10.81	compound	8.54	nmod	11.84	obl	11.12
nmod	9.68	punct	8.33	obl	6.43	nsubj	11.09
advmod	8.49	obj	7.82	nsubj	6.19	nmod	6.70
nsubj	7.91	amod	5.27	amod	4.90	amod	6.61
advcl	5.67	case	5.10	obj	3.93	obj	4.12
obj	3.68	advmod	4.14	xcomp	3.79	xcomp	3.38
mark	3.19	mark	3.86	case	3.45	acl	3.18
compound	3.14	det	3.62	discourse	2.76	conj	2.56

Table 3.9: Deprel frequency, decomposed by arc direction

3.3.1 Algorithms

We follow two approaches to artificially inducing sufficient code-switching. It is important, first, to note that our motivation behind this approach is not to create concrete code-switched examples artificially, but instead to encourage cross-linguistic dependency arc fertility, particularly where cross-linguistic arcs are likeliest to occur. Thus, given the `case` deprel as an example, we sample `case` tokens from the treebank with the language with more closed-class children, swapping out each closed-class token from the other treebank with a certain probability p from these sampled tokens. Algorithm 3.3.1 is an example of how this looks in practice.

Algorithm 1 Generate treebank

procedure GENTREEBANK(T_{src}, T_{cs}, p)
  $sample \leftarrow \emptyset$
  for dep_line in T_{src} **do**
    if $dep\_line.deprel =$ 'case' **then**
      append $dep\_line.form$ to $sample$:
    end if
  end for
  for dep_line in T_{cs} **do**
    if $dep\_line.deprel =$ 'case' **then**
      if $dep\_line.form \notin sample$ **then**
        $dep\_line.form =$ RANDOM($sample$) with some probability p
      end if
    end if
  end for
  return T_{cs}
end procedure

Pragmatically, merely switching out tokens is likely to result in some issues: given how recurrent neural networks function sequentially, it is likely that they would face significant issues with case tokens that are in the wrong order relative

to their heads, for that particular language.

Given that Hindi is a strongly head-final language, and English strongly head-initial, we therefore add an extra step for this pair: we deterministically reorder chunks to ensure that the order of closed-class tokens is reversed for the two. In doing so, we do not bother enforcing the tree constraint: graph-based parsers ought not to worry about rigidity in tree-ness of training data. Algorithm 3.3.1 illustrates how these swaps function.

Algorithm 2 Generate treebank with reordering

```
procedure FixEdits(T_cs, edits)
    for edited_line in edits do
        parent ← edited_line.parent
        Swap(edited_line, parent)
        Swap(edited_line.id, parent.id)
        for dep_line in T_cs do
            if dep_line.parent = edited_line.id then
                dep_line.parent = parent.id
            else if dep_line.parent = parent.id then
                dep_line.parent = edited_line.id
            end if
        end for
    end for
end procedure
procedure GenTreebank(T_src, T_cs, p)
    sample ← ∅
    edits ← ∅
    for dep_line in T_src do
        if dep_line.deprel = 'case' then
            append dep_line.form to sample:
        end if
    end for
    for dep_line in T_cs do
        if dep_line.deprel = 'case' then
            if dep_line.form ∉ sample then
                dep_line.form = Random(sample) with some probability p
                append dep_line to edits
            end if
        end if
    end for
    return T_cs
end procedure
```

3.3.2 Evaluation

Table 3.10 tabulates our results for stochastic switching, with switch probabilities of 0.25, 0.33, 0.50 and 1. As with all parts of our implementation that involve some sort of stochasticity, we set our internal RNG seed to 1337.

1/4	UAS	LAS	wLAS
en-hi	**75.90**	**60.30**	**53.08**
en-hi [reorder]	75.78	60.24	51.96
kpv-ru	65.51	53.42	50.65
1/3	UAS	LAS	wLAS
en-hi	74.96	60.21	52.83
en-hi [reorder]	**76.63**	**60.94**	**53.11**
kpv-ru	65.51	51.74	47.90
1/2	UAS	LAS	wLAS
en-hi	75.78	59.58	52.38
en-hi [reorder]	75.24	60.00	51.61
kpv-ru	64.50	52.63	49.53
Full	UAS	LAS	wLAS
en-hi	71.50	56.21	48.55
en-hi [reorder]	64.16	48.04	45.16
kpv-ru	**67.19**	**54.20**	**51.45**

Table 3.10: F_1 scores for our stochastically generated scrambling system

Two things stand out here: first, we obtain our best-yet Komi/Russian LAS, at 54.20. Further: our scores have virtually no correlation to the proportion of scrambling, as is visible from the fact that our best performing rows are all from tables with different scrambling proportions. This phenomenon seems to indicate that whilst our particular implementation of manual treebank modification was successful to varying extents, the concept itself is not inherently flawed. There are several improvements to our algorithm that we can envision for future work: one being a more rigid form of sampling, where the probability of substitution is not hard-coded, but deterministic, depending on context.

3.4 Network alterations

In this section, we describe a set of alterations that we apply to our baseline neural network architecture to capture code-switching.

3.4.1 Language ID

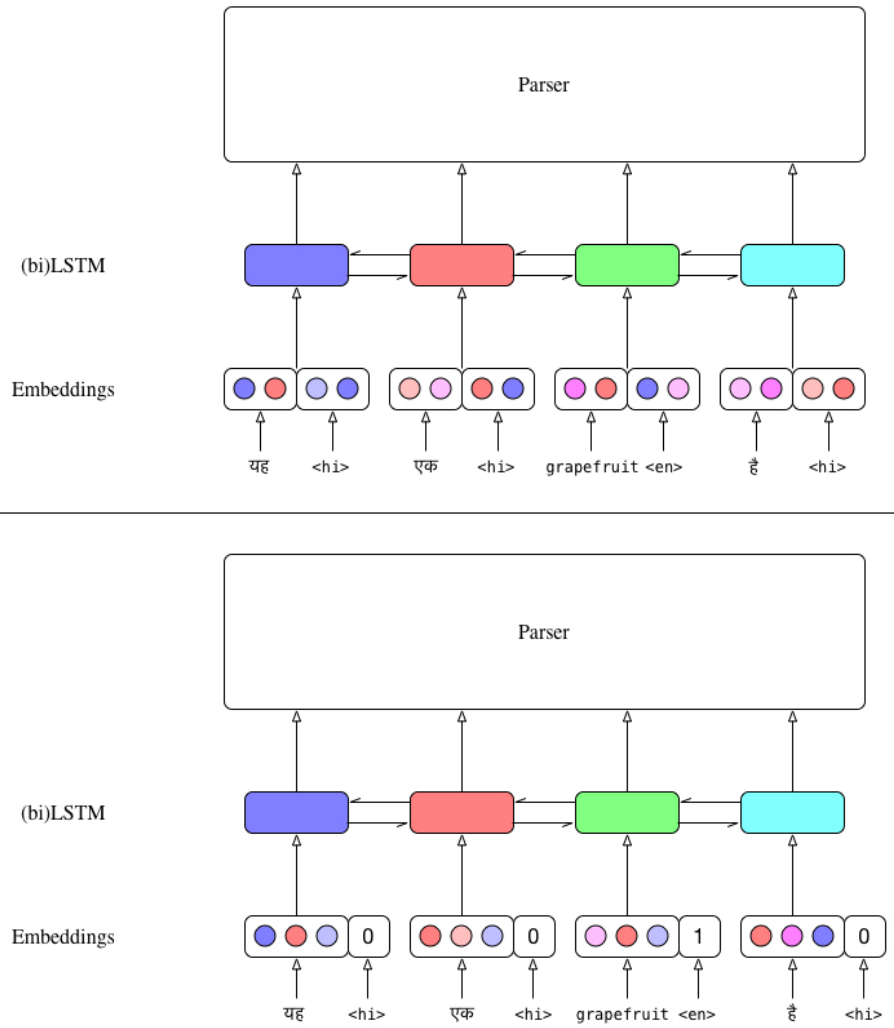

Figure 3.2: Two variable methods to supply language information: dense embeddings and language IDs, for the sentence '*this is a grapefruit*'

The purpose of this section is to introduce a field with the language of the token mentioned, and test whether this influences parsing in any way. We modify our network as shown in Figure 3.2; this allows it to either directly pass the language ID, or to generate (size 100) embeddings for the language ID, which are then concatenated to our word and POS tag embeddings. We compare results with both direct language IDs and the relevant embeddings: the intuition behind using embeddings is that they would capture at least some semblance of context with the way languages are distributed: note that we do *not* have just two language ID tags per treebank, as most treebanks also have separate ID annotations for foreign tokens, or universal tokens like punctuation.

Our motivation for adding language ID is simple: it is plausible that, knowing what language a particular token is in, the parser could 'remember' its knowledge of how that langauge ought to parse, as our training data is also annotated with language tokens. This, in turn, ought to increase performance on parsing chunks in that particular language, as the softmax probabilities are 'biased' in favour of arcs and deprels seen only in that monolingual treebank.

Further, apart from merely using language IDs, we also try to *scramble* our language tokens. This is seemingly the exact opposite of the previous step, however, the intuition behind it is very different. If our parser did not know the language of a particular token - or, more importantly (as it sufficient to not include language IDs at all for it to *not know* the language of a token), think that the language of a token was the *other* language - it would bias the softmax probabilities of arcs across languages. These probabilities are likely to be extremely low, given that they do not exist in training data - our training treebanks, outside of our synthetic treebank experiments, are monolingual, meaning there are code-mixed sentences and therefore no sentences with any sort of dependency relation across tokens in two languages. 'Fooling' the parser into thinking that two tokens from different languages are from the same language offsets this inherently low probability and allows the parser to marginally boost probabilities of arcs where they would otherwise be low.

Evaluation

Tables 3.11 and 3.12 describe our results using language IDs, and language embeddings respectively.

ID	UAS	LAS	wLAS
Gold langid	77.03	61.12	53.48
Scrambled langid	76.97	60.76	52.63
Embeddings	UAS	LAS	wLAS
Gold langid	**77.91**	61.94	53.88
Scrambled langid	76.90	60.94	53.05
(norm) **ID**	UAS	LAS	wLAS
Gold langid	77.09	61.82	53.65
Scrambled langid	**77.91**	**62.22**	**53.95**
(norm) **Embeddings**	UAS	LAS	wLAS
Gold langid	77.27	60.76	51.99
Scrambled langid	76.27	60.91	53.05

Table 3.11: F_1 scores for Hindi/English

ID	UAS	LAS	wLAS
Gold langid	65.29	49.94	46.76
Scrambled langid	65.29	51.96	**48.21**
Embeddings	UAS	LAS	wLAS
Gold langid	65.40	50.28	46.17
Scrambled langid	**66.52**	**52.18**	47.79

Table 3.12: F_1 scores for Komi/Russian

Further, with Hindi, we repeat our experiment with normalised language IDs (normalisation being the deterministic elimination of non-language IDs, such as punctuation).

Discussion

Our language IDs appear to function extremely well: indeed, we obtain our state-of-the-art LAS for Hindi/English with scrambled language IDs. There are, however, several puzzling phenomena that we observe.

1. Without normalisation, Hindi/English language IDs function the exact opposite to Komi/Russian ones: parsing is aided by provided gold standard language IDs rather than scrambled ones. When normalisation is introduced, however, this phenomenon reverses, with scrambled language IDs functioning better for both, indicating that lying to the parser seems to work well.

2. Embeddings appear to work better for unnormalised Hindi/English, and Komi/Russian, whilst direct language IDs work better with normalised Hindi/English. We posit that this difference is purely due to the unpredictability of neural networks, a phenomenon that does not vanish even with consistent hyperparameter initialisation.

It is clear from these results that supplying language IDs does appear to help parsing, at least to the point that further research into the field is warranted. Encoding language embeddings via different neural structures would make for a decent point of departure for further analysis on the theme. A more detailed error analysis is also a valid future direction: one that attempts to investigate why our Hindi/English results flip. One takeaway, however, appears to be both fairly solid, and fairly positive: 'over-annotation' of corpora appears to be unnecessary, and borderline harmful; expanding the language ID space to cover all forms of tokens, ranging from punctuation to foreign words, whilst informative if the downstream is token language identification itself, appears to hurt performance in at least this downstream task.

3.4.2 Multi-task learning

Multi-task learning and dependency parsing

Again, whilst not strictly relevant to the phenomenon of code-switching, we attempt to motivate our use of multi-task learning with our results from another parallel experiment involving jointly learning to parse dependencies, and perform semantic tagging.[3] As this is not related to our main task (but an important side-experiment), we keep descriptions brief.

Semantic tagging [Abzianidze and Bos, 2017, Bjerva et al., 2016] is, fundamentally, the task of assigning (language-independent) semantic categories to words: akin to POS-tagging, but more informative. It is, therefore, similar to other forms of tagging, a sequence-labelling task. By adding this sequence labelling task as an auxiliary task, along with the main task of dependency parsing the English UD treebank, we obtain several improvements in results that we outline in Table 3.13. Our results indicate that sharing every part of the parser network with the semantic tagger network, upto the LSTM layer, provides not-insignificant parsing score gains.

Method	UAS	LAS
Baseline	84.81	80.24
Shared LSTM	85.54	81.03
Dual LSTM	85.81	80.20

Table 3.13: Sharing semantic tagging with dependency parsing; 'dual' LSTMs refer to LSTMs that have both shared and unshared components

These results motivate us to use other sequence tagging tasks, particularly ones that may well be more relevant to parsing code-switched language, as auxiliary tasks to be shared with our main parsing task.

Multi-task learning and code-switching

In this section, we introduce an obvious fit for an auxiliary task: that of, of course, language identification. In spirit, this is very similar to the approach described in the previous section; however, this is a more sophisticated approach to using language ID information to aid dependency parsing.

Similar to Section 3.4.1, we use both regular language tokens and scrambled tokens during training; however, we attempt, in this situation, to instead jointly learn to identify the language of a token. This, fundamentally, reduces to a multi-task learning problem: we share layers (typically embeddings and LSTM layers), and use those to both predict the language of a token, and to learn to predict dependency relations.

The intuition behind this approach is similar to that in Section 3.4.1. However, we invert the problem: instead of having the language ID as an actual feature input to the parser, we make it a feature that the parser is required to simultaneously predict. The intuition works both ways: being able to predict the language

[3]This research was also jointly conducted with Mostafa Abdou and Artur Kulmizev, and is (at the time of this submission) under review at EMNLP-2018

of a token ought to improve parsing performance on chunks in that token's language, whilst *not* being able to predict the language (or, rather, being able to predict the wrong language) improves parsing performance on cross-linguistic chunks.

Implementation

Predicting language IDs is a simple sequence prediction task, akin to the well-known task of POS tagging, albeit with a significantly smaller set of output labels. Interestingly, preliminary experiments on our development data (that we do not repeat here, for brevity) indicate two things:

- Unlike with our experiments semantic tagging, sharing the LSTM winds up hurting downstream performance; the most 'ideal' sharing setting is hard sharing of the embeddings layer, which modify the word representations directly, without modifying any higher-level abstractions of these words.
- Our multi-layer perceptron, that attempts to predict language tags, functions significantly better with three layers (two non-linearity introducing layers and one output layer) than it does with two.

Figure 3.3 is a diagram showing a concrete implementation of this network; Table 3.14 describes our hyperparameters for the language identification part of the network; 'input' is in quotes as our input layer is, of course, not a generic input layer, but the first unshared layer for this particular downstream task. We use standard cross-entropy loss after calculating (log) softmaxes over our output layer.

Parameter	**Value**
'Input' layer	150
Hidden layer	100
Dropout	33%
Activation fn.	ReLU
Learning rate	$1 * 10^{-3}$

Table 3.14: Hyperparameters for the language prediction section of our network

Evaluation

Note that unlike in the previous experiment, we do not really use language embeddings: this is quite pointless, as prediction tasks typically use one-hot vectors as target outputs. Further, note that all our language IDs are inherently normalised: during test time, we do not care about the language of the token, as language IDs are only useful for learning our internal parameters during *training*. Thus, this method is less 'needy' than the previous, that requires language IDs as an input feature at test time. Our results are tabulated in Table 3.15.

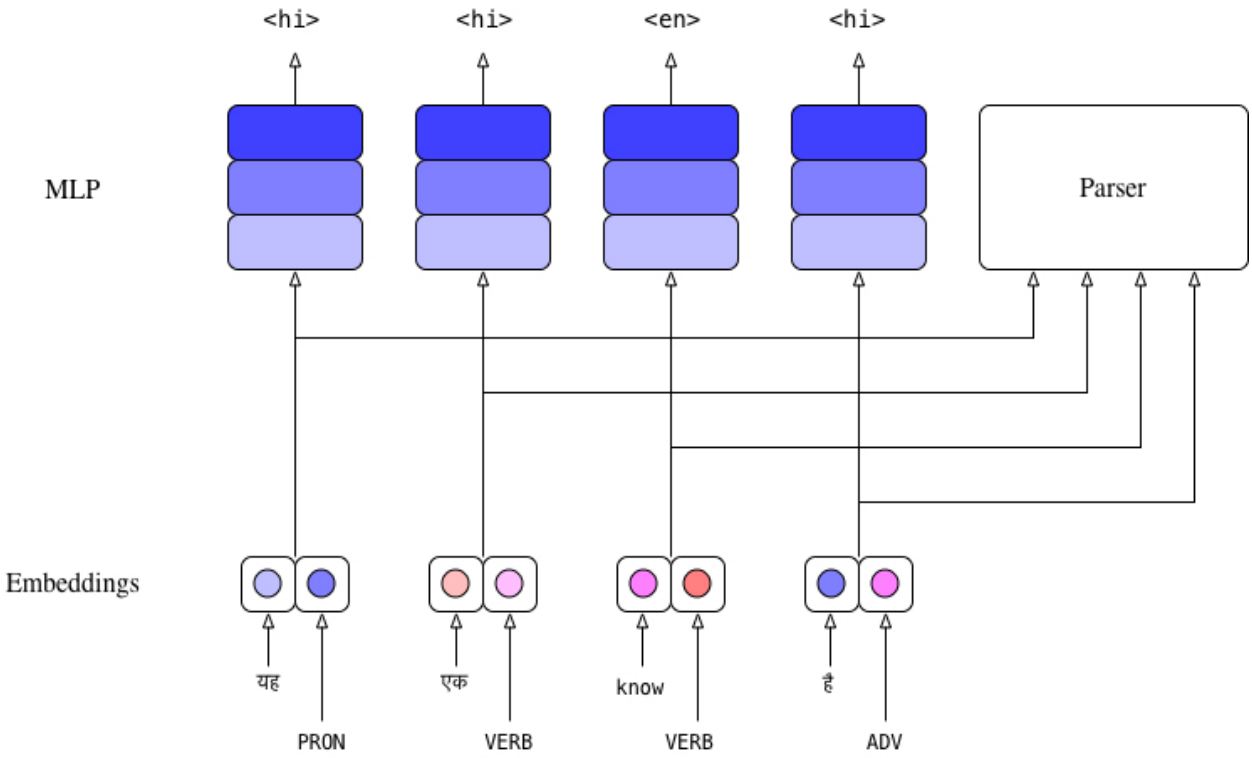

Figure 3.3: Architectural block diagram of our MTL system; the output of our embeddings layer propagates both to our parser and to a multi-layer perceptron that attempts to predict language

en-hi	UAS	LAS	wLAS
Gold langid	**77.60**	**61.12**	**52.66**
Scrambled langid	76.87	60.73	52.58
ru-kpv	UAS	LAS	wLAS
Gold langid	65.40	50.28	46.17
Scrambled langid	**65.85**	**52.07**	**48.71**

Table 3.15: F_1 scores for language IDs supplied as embeddings

Discussion

This section also shows the same rather curious disparity in performance as the previous: Komi/Russian seems to perform *better* on all metrics when supplied with scrambled language IDs (albeit with less stark a difference here), whilst Hindi/English performs better with 'honest' language IDs. The replicability of this result indicates that it cannot be merely a coincidence: the phenomenon clearly seems like an interesting avenue for future study.

3.4.3 Domain shift

An interesting way to treat the problem of dependency parsing code switched language is to picture code-switched language as a shift in domain from regular language. There has been considerable prior work on handling domain shift [Ruder and Plank, 2018, Ganin et al., 2015, Pei et al., 2018] we base our approach largely on handling domain shift via backpropagation, *à la* Ganin and Lempitsky [2014].

Put succinctly, the idea behind representing domain shift with backpropagation is simple: by making the network incapable of predicting the domain of a particular sentence, the network begins to learn representations that are domain-invariant. Thus, in principle, this involves *unlearning* the domain of a sentence, which is fundamentally a tag stating whether the sentence is monolingual or multilingual. Whilst in theory we could reduce our sentence-level domains to L_1/L_2/multilingual domains, this is unnecessary as both L1 and L2 are 'foreign' domains that we do not care about.

Architecture

Concretely, this is implemented by adding a domain prediction network (similar to the auxiliary setting in Section 3.4.2). Unlike in Section 3.4.2, our loss function does not add the label prediction loss, but *subtracts* a scaled variant of this loss. Figure 3.4 describes the architecture of our system.

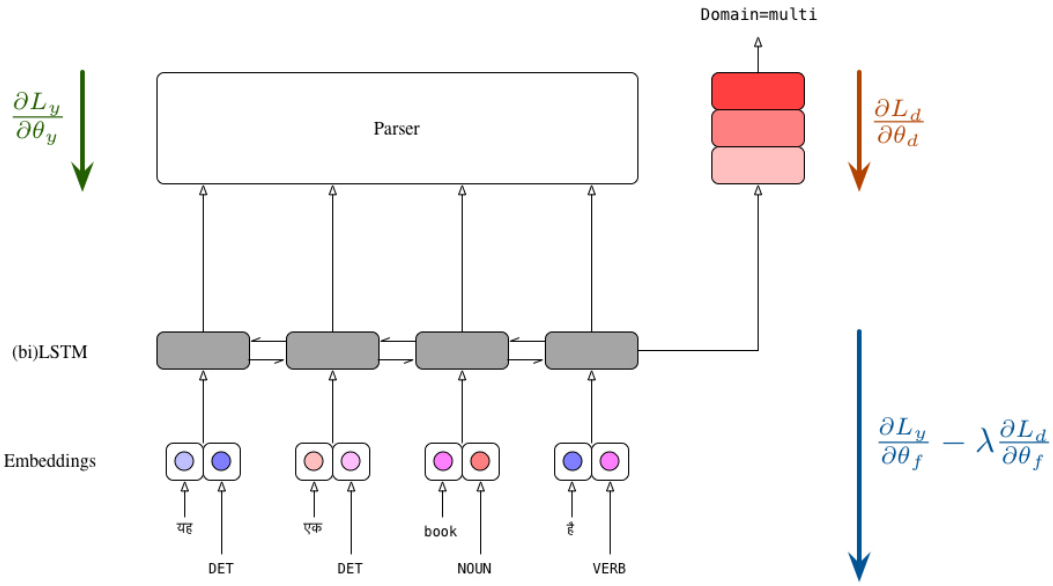

Figure 3.4: Domain prediction system; arrows indicate direction of backpropagation

More formally, given a set of parameters θ_y for the unshared parts of our dependency parser, a set of parameters θ_d for the unshared parts of our domain predictor, and a set of parameters θ_f for the shared parts of both subnetworks, L^i subscripted appropriately for the appropriate loss for example i, and assuming stochastic gradient descent as our optimiser, we define our backpropagation step as:

$$\theta_f \leftarrow \theta_f - \mu \left(\frac{\partial L_y^i}{\partial \theta_f} - \lambda \frac{\partial L_d^i}{\partial \theta_f} \right)$$
$$\theta_y \leftarrow \theta_y - \mu \frac{\partial L_y^i}{\partial \theta_y}$$
$$\theta_d \leftarrow \theta_d - \mu \frac{\partial L_d^i}{\partial \theta_d}$$

Whilst the unshared parts of the network clearly follow standard stochastic gradient descent (with a learning rate μ), the interesting part is the gradient descent step for the shared layers; note two interesting facts:

1. We *subtract* the loss from the domain classifier from the total loss (and therefore add it to the gradient step). The reason for this is obvious: we do *not* want our parser to be able to learn to tell domains apart. This is, in spirit, similar to shuffling language tokens, albeit more rigid.

2. We scale our domain classification loss down by a factor λ. This is essential, as an unscaled classification loss would result in our network merely using two disparate representation spaces for both domains. Practically, we follow similar hyperparameter principles for λ as in the paper, and gradually change its value from 0 to 1, following the formula:

$$\lambda = \frac{2}{1 + \mathrm{e}^{-\gamma p}} - 1$$

where p is the fraction of the current batch that has been processed, and γ is a scaling factor (set to 10).

Augmentation with dev

This is the first of our adapatations that involves augmenting our training data with development data.

Whilst this is slightly regrettable - as dev data ought to be used solely for hyperparameter optimisation - it makes sense to see how 'far' it is possible to get when training data is augmented with a minimal number of annotated sentences from the relevant domain, i.e. code-switched sentences. 'Dev' data, in this context, is a bit of misnomer - what was once our 'dev' data is now a part of our training data, and we refer to it as such hereafter.

	UAS	**LAS**	**wLAS**
en-hi	79.54	65.58	57.13
kpv-ru	67.54	53.11	49.85

Table 3.16: Baselines for standard parsing with additional dev data

It is clear that there ought to be separate baselines, given this augmentation - it is a bit unfair to compare our methods with our naive baselines, given that our baselines do not have all this extra data, which, even if minimal, is informative.

We therefore create a new set of baselines *specifically* to compare these experiments to: this new baseline is evaluated fundamentally the same way as our old baselines; we evaluate a parser trained on a concatenation of both monolingual treebanks and the multilingual 'development' data. These results are outlined in Table 3.16, and are relevant for both this section and the next.

Domain specification

As mentioned above, we annotate our domains two ways - at a word level, and at a sentence level. Domain shift via backpropagation requires data from both domains during train time, though these domains may differ in distribution during test. We therefore use our multilingual training data as one class, and our monolingual as another.

At a word level, our labelling is fundamentally the same as in Section 3.4.2. We outline our results in Table 3.17.

	UAS	**LAS**	**wLAS**
en-hi	69.26	49.35	40.19
kpv-ru	67.05	50.33	46.88

Table 3.17: Results for our domain shift system

Discussion

Unfortunately, our results massively underperform basically every other system that we have evaluated. This hints to - even if it does not confirm - the fact that code-switching *cannot* necessarily be seen as a distinct domain to monolingual text. Why this is the case is curious: we propose that this is due to domain being a sentence level phenomenon, which is insufficient to modify representations of *words* within the sentence, to better represent code-switching. Further experiments are, however, necessary before a solid conclusion can be drawn; an area of interest, if our hypothesis holds true, would be to evaluate *other* code-switching tasks using sentence-level representations [Kiros et al., 2015, Logeswaran and Lee, 2018, Conneau et al., 2017a], which are fundamentally an extension of the principle of word embeddings to complete sentences. They have been used in several downstream tasks in the past, many of which have been gathered into evaluation sets, such as SentEval [Conneau and Kiela, 2018]; we propose generating similar datasets for code-switched language as an initial step towards evaluating the effects of domain shift on sentence representations over code-switched data.

We also suggest experimenting with different values of λ, on the (minor) offchance that the scaling method described in the paper does not apply to our specific problem.

3.4.4 Development weight learning

Architecture

In this section, we outline the most 'interesting' of our network modifications, one that we call dev-weight learning; this, to the best of our knowledge, is an architecture that has not been used in any form of NLP task in the past, and we hope to motivate it and its use in other NLP tasks, as relevant.

The name 'development' weight learning is a bit of a misnomer; fundamentally, the essence of what we propose is a system that can to more effectively parse code-switched data, given some additional 'development' data (which, in real-world scenarios, would be considered small amounts of additional training data). Our system works by introducing, along with our standard final head prediction softmax-value matrix, an additional weight matrix that, essentially, learns weights that are element-wise multiplied to the softmax predictions, in order to have them more realistically resemble code-switched data.

The motivation behind this being an improvement over merely learning using code-switched data is the relative simplicity of the model: by shifting our weights slightly above or below 1 (our initial value for every weight element), our softmax probabilities are also altered. The architecture of our model is presented in Figure 3.5.

Fundamentally, our algorithm works in three steps:

- Initially, we learn from training data, similar to in our original model. The stacked MLP outputs from our original parser (represented by $H^{(dep)}$ and $H^{(head)}$ here), however, are passed through *two* biaffine layers with independent internal parameters; the first of these attempts to predict the softmax probabilities of the heads, whilst the second attempts to shift all its internal parameters to generate a weights matrix with all 1s. Over every epoch in our training phase, the output of this layer is moved closer to predicting all ones. At this phase, the Hadamard product represented by $S^{(arc)} \circ W^{(train)} \approx S^{(arc)}$ Note that while teaching our second biaffine layer, we freeze all network parameters except the parameters of the layer itself. We are not attempting some form of multitask learning; indeed, the idea that learning to predict a matrix where every element is 1 can lead to representations that improve dependency parsing is a bit silly.

- Next, we learn from our development data. This step involves learning weights that correspond more to the actual dependencies being predicted during the development phase. As such, we replace our all-ones matrix with a matrix where existing dependency relations are indicated by 1s, and others by 0s: in theory. In practice, however, we find that using 0s for all non-existent arcs winds up having our network rapidly converge to obtain 0s for some weight values; this is unacceptable, as it causes problems with our post-Hadamard softmax, leading to cycles within sentences. We therefore either set either all 0s to a value $\lambda < 1$, or set all 1s to a value $\lambda > 1$ (and simultaneously set all 0s to 1).

  Learning from dev data thus allows our weights matrix to shift from being a matrix that is approximately all ones, to being a matrix that is still full of

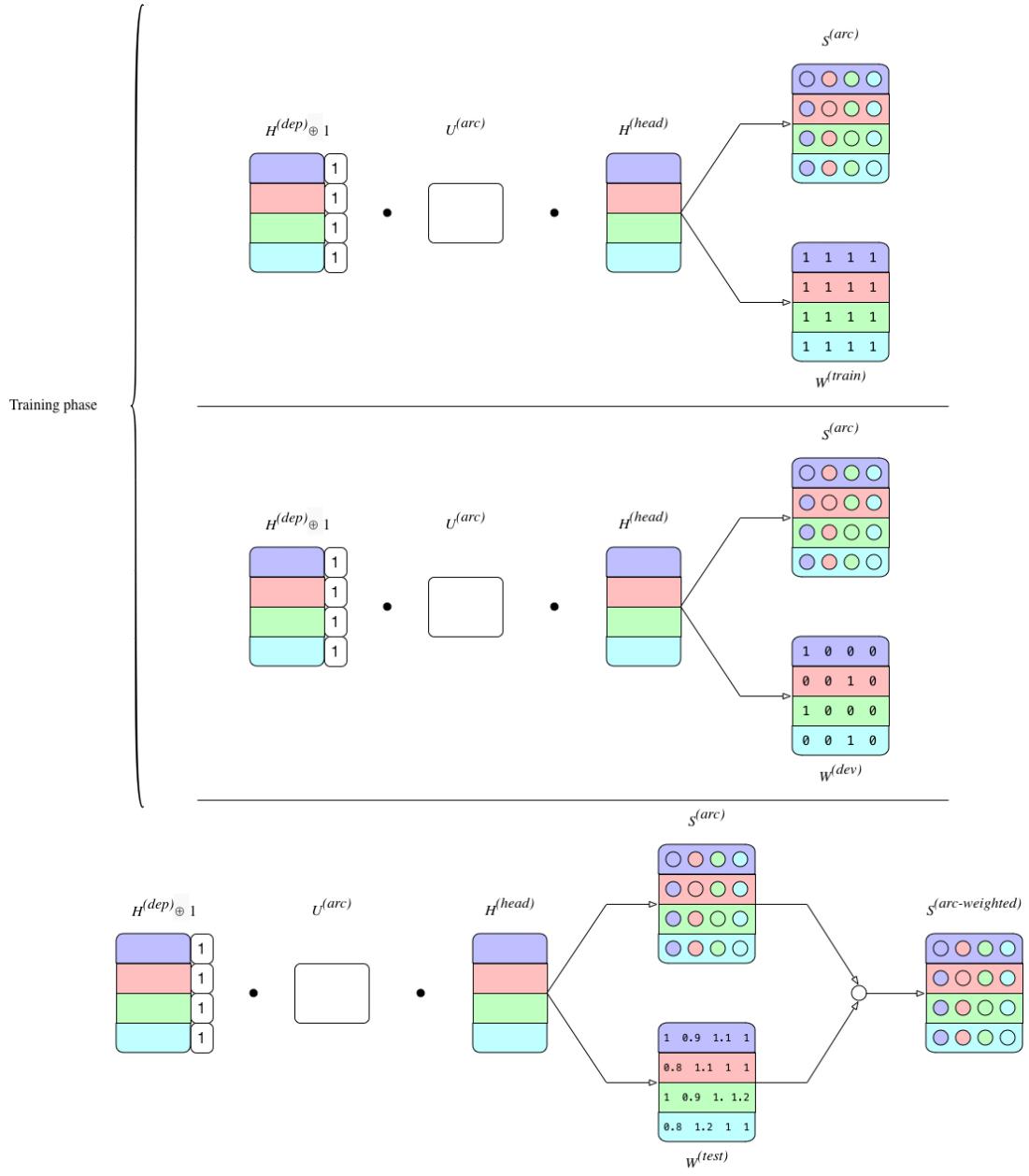

Figure 3.5: Learning from development data in three steps

values close to one: but further away than earlier, and more 'informatively' divergent.

- Finally, during our prediction phase on unseen test data, we use our parameters that we attempted to train to the point of usefulness, on development data, to predict a weights layer relevant to our current sentence. Our intuition is that despite *never having seen* the softmax probability matrix $S^{(arc)}$, applying some sort of composition function $f(S^{(arc)}, W^{(test)})$ ought to, without any learning involved, result in a weighted softmax prediction that more adequately captures code-switching: the likelihoods are, fundamentally, weighted *up* by seen code-switching data.

Concretely, we use the same hyperparameters for learning $W^{(train)}$ that we do for learning $S^{(arc)}$ in our original parser ('same' implying similar dimensions and initialisations, not them being shared). Note that we use a mean-squared error loss instead of our standard cross-entropy loss.

Evaluation and results

For our λ values, we perform two experiments. One involves gradually increasing the value of λ from 0 to 1; we use the same increase method described in 3.4.3, albeit without a scaling γ (due to our test being too small to justify it), given by:

$$\lambda = \frac{2}{1 + \mathrm{e}^{-p}} - 1$$

where p represents our training progress as a fraction of the current batch.

Evaluation

Our dev-learning results are not necessarily the most promising: we present them in Table 3.18. Note that these results also ought to be compared to our baselines from Section 3.4.3; we are, after all, using dev data in training.

We evaluate our system with three settings: one with our weights attempting to backpropagate existing deprels to 1 and non-existent ones to 0.5; one attempting to backpropagate existing deprels to 1.5 and non-existent ones to 1; and finally, one fixing existing deprels to 1 and scaling up the non-existent deprels from 0 to 1.

Our 0.5 setting did not result in valid CoNLL-U files for Hindi/English; this implies that the multiplication by 0.5 resulted in some softmax weights taking on values close to 0, thus confusing our spanning tree algorithm.

Discussion

We believe that there is significant potential in improving on this method; there are several modifications that could be made to this architecture to help it to learn better, that we could not evaluate due to time constraints. For instance, the system could literally just learn the additional matrix via backpropagation, evaluating the gradients by backpropagating through the Hadamard product every step:

0.5	**UAS**	**LAS**	**wLAS**
en-hi	-	-	-
ru-kpv	61.15	48.20	45.43
1.5	**UAS**	**LAS**	**wLAS**
en-hi	**77.60**	**62.19**	**54.37**
ru-kpv	**66.89**	**51.97**	**48.93**
Variable	**UAS**	**LAS**	**wLAS**
en-hi	73.54	58.21	50.62
ru-kpv	62.95	49.67	46.53

Table 3.18: Development learning results for different values of λ

$$\frac{\partial(X \circ Y)}{\partial\theta} = Y\frac{\partial X}{\partial\theta} \circ X\frac{\partial Y}{\partial\theta}$$

Our experimental results also clearly indicate that *overweighting* the existing dependency arcs, rather than underweighting non-existing ones, performs significantly better. This has several implications, the first and most obvious one being the fact that we need to spend more time on hyperparameter optimisation, 1.5 being an arbitrary setting. More concretely, it might make sense to introduce some sort of scaling value, that instead of scaling up from 0 to 1, does so from 1 to 2.

Finally, regarding the actual success of the method itself: an interesting future avenue for evaluation is with domain adaptation itself.

4. Predicting code-switch points

An additional task that we introduce as part of this thesis is the task of predicting code-switch points; we motivate this with several potential directions for future research that can make use of systems that perform well at this task.

It is crucial to define at this point precisely what a code-switch point really is: fundamentally, we define a code-switch point as any point in a running stream of text where the *next* token is going to be from a different language.

4.1 Background

The task of predicting code-switching points is not entirely novel; one of the first papers to adequately tackle this topic, for Spanish-English, was Solorio and Liu [2008a], which is far from being a 'modern' paper; one of the disadvantages of this paper is their relatively complex feature function, something that was understandably necessary in 2008, prior to the advent of neural networks. Papalexakis et al. [2014] also attempt to predict code-switching points in a Dutch/Turkish corpus: they use an even more complex set of features, often boolean, such as the presence/absence of emojis in a trigram sequence. There are, to the best of our knowledge, no modern papers that attempt to predict code-switching points using any form of neural architecture.

It quickly becomes very intuitive to describe this task as a sequence prediction task, akin to, as we mentioned earlier, POS tagging, and the task of predicting language IDs for the *current* token that we described in Chapter 3. One extremely significant difference is, however, is the fact that this is a *time-shifted* sequence prediction task: we are required to predict the 'label' for the *next* token, given information from the current token and its tag history, along with some form of sentence model built from preceding/succeeding tokens.

4.2 Evaluation

Evaluation metrics for this particular task are by no means consistent: Solorio and Liu [2008a], for instance, use human judgements as an evaluation metric. The unfortunate side-effect of this lack of consistency is that our results are not directly comparable to others; however, we hope to establish these results as, if not a competitive system, at least a significant baseline for future efforts.

We define our evaluation metric on the basis of two factors: simplicity and motivation. Specifically, our metric derives from our definition of the task as a time-shifted sequence prediction task: we merely evaluate the number of language tags (which, fundamentally, act as a proxy for the boolean 'switch occurred' tag) that have been predicted correctly, akin to, for instance, POS tagging.

4.2.1 Motivation

It is important, at this point, to take a moment to describe precisely what the point of this task is, and how we envision it being used in the future. Our goal

is to predict code-switching points specifically from the perspective of being able to 'easily' generate code-switched corpora from word-aligned parallel corpora. To motivate the validity of this method, we make two assumptions:

1. The information conveyed by parallel sentences in multiple languages is the same; this is a fairly obvious assumption.

2. For all parallel sentences in languages L_1 and L_2 that convey the same information, there exists a sentence that can be constructed using non-zero length chunks from both L_1 and L_2, that conveys the same information as the L_1 and L_2 equivalents.

From these assumptions, we posit that it ought to be possible, given certain language pairs, to generate a multilingual sentence that not only conveys the same information as the monolingual sentences, but is, further, 'grammatical', according to the intrinsic grammar of all multilingual sentences over $L_1 \cup L_2$.

We propose the use of a predicted code-switching point to impose this additional grammaticality construct. Whilst by no means deterministic, we posit that our prediction, combined with word-alignments, ought to allow for adequate construction of code-mixed sentences.

4.3 Pre-processing and data

For our treebanks tagged with multiple language IDs, including eg. special tags for punctuations and foreign words, we experiment with two settings, one where we also try to model predictions for these tags, and another where we do not: we assign each extra, non L1/L2 token the language tag of the previous token with a language associated with it, or the next token with a language associated with it, if the token to be re-tagged is word-initial. We evaluated our systems on both these models.

Note that this task explicitly does require annotated code-switched data for training (although just POS annotations are sufficient, for the most part). We therefore use the Hindi/English dataset annotated (both with dependency relations and with POS tags) by Bhat et al. [2018]. Statistics for both these corpora are provided in Table 4.1; note that these are statistics for all three folds (train + test + dev) for German/Turkish.

Metric	**en-hi**	**de-tr**
Sentences	1309	1029
Words	18K	17K

Table 4.1: Naive corpus statistics for both our training corpora

For Russian-Komi, on the other hand, we do not have sufficient data to train any sort of functional system for this task: indeed, the size of the corpus (40 sentences) is the same as our batch size for other tasks. In order to introduce a more meaningful language pair, therefore, we refer the reader to a Twitter-based, POS annotated German/Turkish code-switching corpus [Çetinoğlu and Çöltekin,

2016]. We present more interesting corpus statistics for both these corpora (the extra Hindi/English corpus, and the German/Turkish corpus) in Table 4.3. We omit cross-linguistic arc fertility as it is irrelevant to this task.

	en-hi		**de-tr**	
	Orig. train	Norm. train	Orig. train	Norm. train
M-index	0.44	1.00	0.16	0.16
Language entropy	1.66	1.00	1.83	0.76
I-index	0.41	0.22	0.43	0.11
Burstiness	0.01	0.16	-0.20	0.03
Span entropy	1.40	1.47	0.58	0.61

Table 4.2: Code-switching statistics for our treebanks; note that Komi/Russian have only a single unified, normalised test treebank

	de-tr			
	Orig. dev	Norm. dev	Orig. test	Norm. test
M-index	0.24	0.64	0.23	0.61
Language entropy	1.90	0.82	1.77	0.81
I-index	0.47	0.14	0.40	0.12
Burstiness	-0.63	0.23	-0.08	0.18
Span entropy	0.23	1.88	0.67	0.63

Table 4.3: Code-switching statistics for our treebanks; note that Komi/Russian have only a single unified, normalised test treebank

4.4 Evaluation

We evaluate multiple architectures on our task. Note that as the task involves predicting the probability of a code-switch at a future token, we cannot run our language IDs into a bidirectional LSTM: indeed, in the sentence-composition domain we envisage this system being used, we would not even have access to 'accurate' future language IDs: we (initially) run our system on monolingual sentences, which means that future language IDs would be meaningless information. This results in several architectures that we compare:

- No information on current token language (Figure 4.1)
- Single unidirectional LSTM for form, POS tag and language ID (Figure 4.1)
- Bidirectional LSTM for form and POS tag, unidirectional LSTM for language ID (Figure 4.2)
- Bidirectional LSMT for form and POS tag, direct language ID embedding (Figure 4.2)

Our architectures are thus conceptually very similar to previous architectures we have used throughout this work for sequence labelling tasks. We describe our hyperparameters in Table

Parameter	Value
Embed dim.	100
LSTM dim.	100
LSTM layers	3
Input MLP dim.	200
Hidden MLP dim.	100
LSTM dropout	50%
Dropout rate	33%
Learning rate	$1 * 10^{-5}$

Table 4.4: Hyperparameters for our code-switch predictor

System	en-hi		de-tr	
	Raw	Normalised	Raw	Normalised
No lang.	41.4	43.41	**50.58**	73.25
(uni)LSTM	36.65	7.87	2.47	5.72
(uni + bi)LSTM	28.95	50.20	50.59	21.03
(bi)LSTM + embed	**57.39**	**70.45**	47.80	**83.1**

Table 4.5: Results for our four systems for code-switch prediction

4.5 Analysis

One thing immediately stands out from these results of ours: the fact that unidirectional LSTMs wind up ruining performance, to the point that without some form bidirectional LSTM in our architecture (model 2), it just completely fails at parsing our data, except for unnormalised Hindi/English. We suggest that this is due to the specific initial internal parameters based on our seed, and not representative of the system in general.

Having accepted the unreasonable effectiveness of bidirectional LSTMs, the next that is visible is the usefulness of language ID labels, particularly post normalisation, where gains are more significant than without normalisation, to the point that our German/Turkish model without language IDs outperforms the one with.

Whilst there are numerous architectures that can be evaluated in this scenario: one possible alternative is using auxiliary losses, *à la* Plank et al. [2016], or even experimenting with multi-task learning settings as we did in our previous task. There are a vast number of potential architectures that we should have liked to evaluate but could not due to time constraints; we take this opportunity to provide our best results for our architectures, however, in the hope that they serve as a baseline for future efforts with this task, whether as independent research or even as a shared task at a conference.

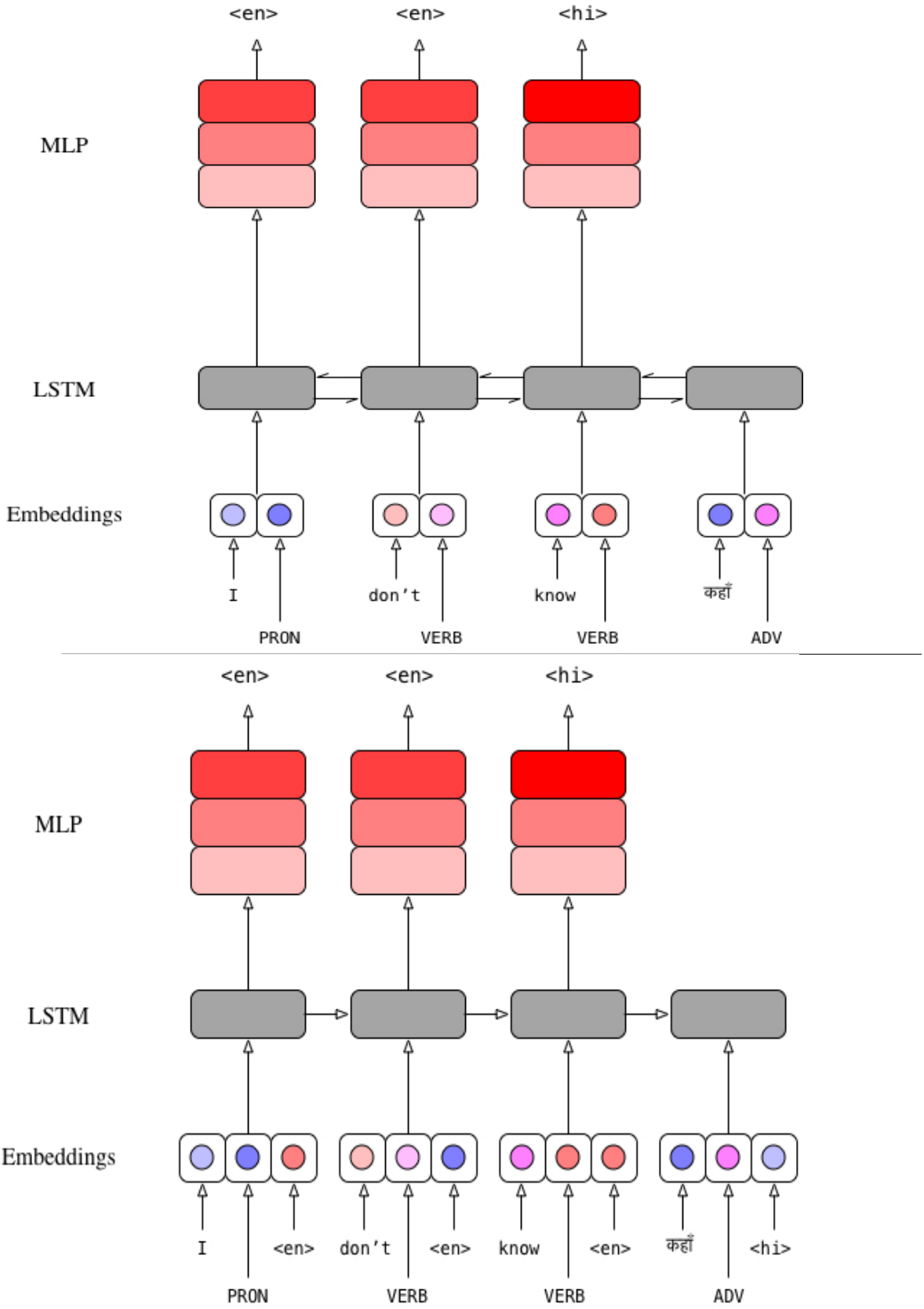

Figure 4.1: Architectures for supplying no language information, and unidirectional LSTMs

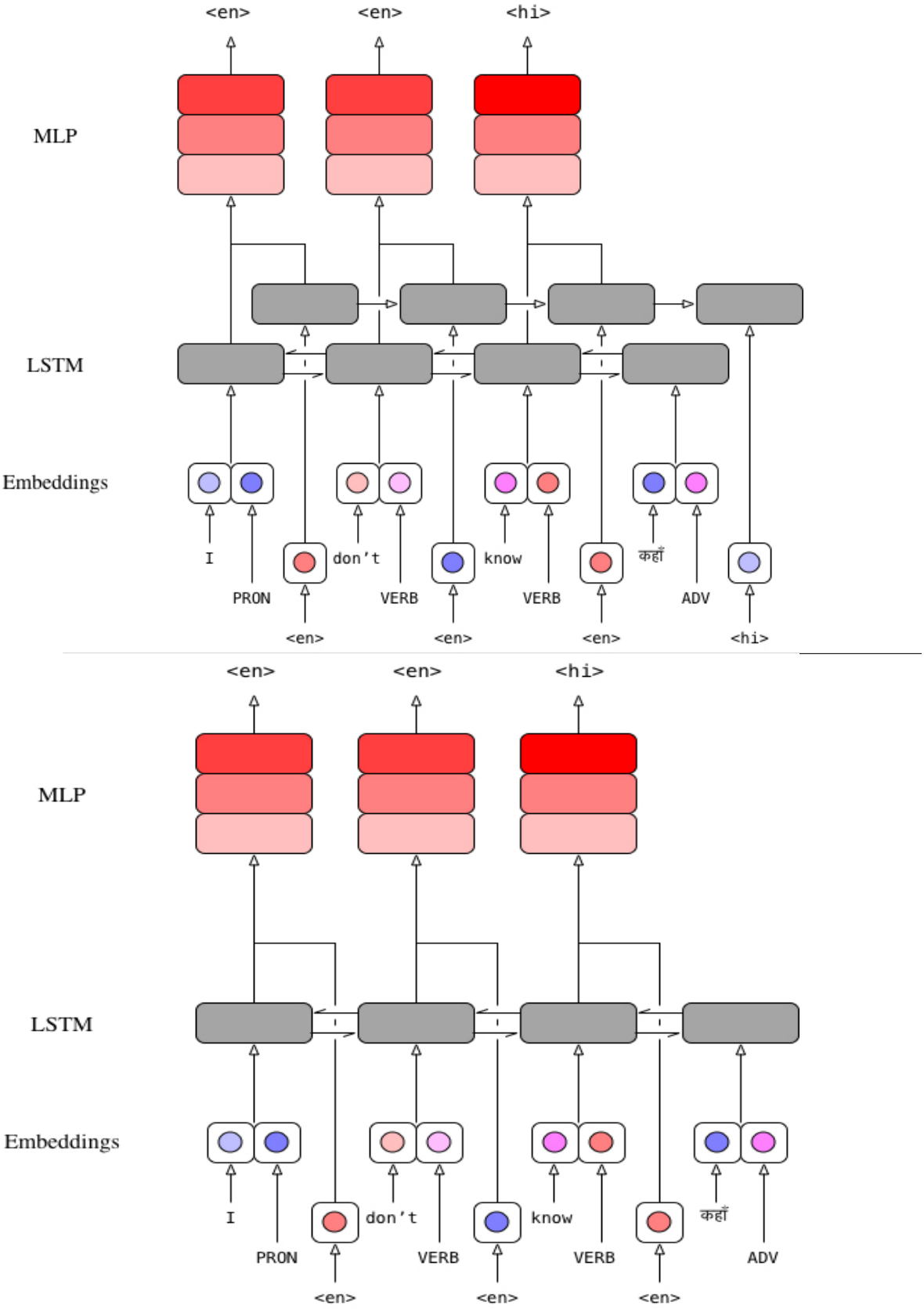

Figure 4.2: Uni+biLSTMs, and biLSTMs + embeddings

Conclusion

Results

For brevity, we summarise all our *best* results obtained with each method in a single table here, before proceeding to analyse it and draw conclusions. We define our 'best' result for a particular set of experiments as the result with the best LAS: realistically, this is the most important score that we have, as far as most downstream tasks are concerned. These results are tabulated in Table 4.6.

en-hi	**UAS**	**LAS**	**wLAS**
Baseline	76.05	60.15	52.27
1/3 + reorder	76.63	60.94	53.11
Compressed + concat.	76.94	61.61	53.81
Scrambled langid + norm.	**77.91**	**62.22**	**53.95**
MTL + gold	77.60	61.12	52.66
Baseline with dev.	**79.54**	**65.58**	**57.13**
Domain shift	69.26	49.35	40.19
Weight learning	77.60	62.19	54.37
kpv-ru	**UAS**	**LAS**	**wLAS**
Baseline + char	65.40	53.08	49.97
1	67.19	**54.20**	**51.45**
Compress + concat. + char	**67.97**	53.98	50.99
Scrambled langid.	66.52	52.18	47.79
MTL scrambled	65.85	52.07	48.71
Baseline with dev.	**67.54**	**53.11**	**49.85**
Domain shift	67.05	50.33	46.88
Weight learning	66.89	51.97	48.91

Table 4.6: Brief descriptions per experiment for the configuration with the best relevant result

We compare these to the original results obtained by the two original papers we based our work on, in Table 4.7.

Paper	**UAS**	**LAS**
(**en-hi**) Bhat et al. [2017]	74.40	64.11
(**kpv-ru**) Partanen et al. [2018]	66.32	53.89

Table 4.7: Competitive results; Komi results are the weighted average of the two splits presented in the original paper

Whilst we do not have the state-of-the-art LAS for Hindi/English, this is easily explained by the fact that our parser architecture likely performs relatively poorly

at recalling deprels; this says nothing, of course, about our augmentation methods. It is harder to directly compare our Komi/Russian results to the original, due to the fact that the authors decided to evaluate separately per constituent treebank; however, weighting their independent results by the corresponding corpus sizes and averaging them out gives us their combined scores, that we manage to beat both on UAS and on LAS.

We would, nonetheless, like to emphasise that, whilst an excellent additional bonus, beating other people's baselines was never our original goal, as that would have involved spending significant amounts of time on our baseline parser architecture. Our goal has always been to beat our *own* baselines, which, as Table 4.6 demonstrates, we clearly did.

Discussion

Our results without dev augmentation are extremely encouraging: virtually all of them outperform our baseline. There are several tangible takeaways that we would like to mention, based on these results:

- There is no one-size-fits-all solution that works with both languages. Whilst we would have loved to have more annotated code-switched dependency treebanks to evaluate this hypothesis more rigidly, we feel fairly confident in making this particular call. In particular, the biggest difference between our Hindi/English and Komi/Russian treebanks was the relative sizes of the monolingual treebanks, and other data: whilst Hindi is not the best-resourced language, Komi is relatively extremely underresourced, and there is thus an inherent disparity in parsing results on both.

- Focusing excessively on parser architectures often results in neglecting training data: often, it is quite possible to obtain improved results by – instead of fixating on the best architecture – thinking about the data instead. Indeed, the fact that our relatively unsophisticated token swapping method is our state-of-the-art for Komi indicates that it might, perhaps, be easier to improve performance in a wide-range of code-switching tasks (not necessarily just dependency parsing) by attempting to generate code-switched data with monolingual data.

- Embedding mapping systems are not necessarily as groundbreaking as they seem, for languages that differ syntactically, or for languages that are underresourced; indeed, this is fundamentally just a reiteration of Søgaard et al.'s [2018] similar claim, which we posit holds true even with different embedding mapping systems, for completely different downstream tasks.

- Language information always helps, whether scrambled or gold-standard. Whilst the fact that gold-standard language data helps sometimes whilst scrambled language data helps at other times might seem a bit strange, we offer a perfectly logical explanation for this: our gold-standard language tokens work best in a multi-task learning setting; we believe that this is due to the fact that learning the language of a token simultaneously, despite pushing the two vector spaces apart, provides more structure to them by

constraining their potential variance with language as a 'clamping factor' during training.

- Perhaps a naive approach to learning from additional domain-relevant data does indeed work best: our baseline for our system augmented with development data significantly outperforms the alternatives, providing a strong motivation for simplicity above all else.

Research questions

Relevant to the research questions that we posed at the start of this work, therefore: the answer to *whether* there exist techniques by means of which baseline results can be optimised is clearly positive. Designing, evaluating and describing some of these potential techniques is a key part of this thesis, and results for many of these experiments are positive, while some fail.

As to the more difficult second question – whether our models, trained on large amounts of monolingual data, can beat models trained on smaller code-switched corpora – we present in Table 4.8 the results obtained by Bhat et al. [2018] for their parser, trained on the training set we described in Chapter 4. These are (perhaps understandably) significantly better than any of our results, which forces us to conclude that, unfortunately, there is no way to improve parsing results over training on actual, annotated code-switched data.

UAS	**LAS**
82.73	73.38

Table 4.8: Results for parsing Hindi/English when trained on an actual training split

And finally, we draw the reader's attention to the task on predicting code-switch points: clearly, this is something that is absolutely possible. Whilst we do not claim to have reached an 'unbeatable' state-of-the-art, our results could serve as a starting point for future research in the domain.

Future work

Whilst not a focus of this thesis, it quickly becomes extremely clear how annotated evaluation sets for code-switched data are essential, for future work in the domain. Annotating some of the many existing POS-tagged corpora with dependency labels is an excellent way to supplement existing treebanks and encourage research in the field.

One of the key takeaways of this thesis – the fact that manually altering monolingual data to generate code-switched data – needs to be studied in more rigid detail, with linguistic and grammatical insights. Whilst simulating code-switched corpora has been done before [Wick et al., 2015], these are relatively naive and rely on uninformed random sampling – as our results show, reordering chunks to simulate linguistic properties such as head directionality can help significantly.

Finally, there are several exciting possibilities with the task of predicting code-switch points; as we have already mentioned, generating code-switched corpora based on parallel corpora is one of these. We intend to continue with research in this direction, and attempt to evaluate the more interesting downstream task of generating these corpora.

Apart from these suggestions, that we believe are high priority, most other task-specific future avenues of research have already been mentioned in the relevant sections for the specific tasks.

Bibliography

Lasha Abzianidze and Johan Bos. Towards universal semantic tagging. In *Proceedings of the 12th International Conference on Computational Semantics (IWCS 2017) – Short Papers*, pages 1–6, Montpellier, France, 2017.

Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972. ISBN 978-0-13-914556-8.

Fahad AlGhamdi, Giovanni Molina, Mona Diab, Thamar Solorio, Abdelati Hawwari, Victor Soto, and Julia Hirschberg. Part of Speech Tagging for Code Switched Data. In *Proceedings of the Second Workshop on Computational Approaches to Code Switching*, pages 98–107, Austin, Texas, November 2016. Association for Computational Linguistics. URL `http://aclweb.org/anthology/W16-5812`.

Waleed Ammar, George Mulcaire, Miguel Ballesteros, Chris Dyer, and Noah A. Smith. Many Languages, One Parser. *arXiv:1602.01595 [cs]*, February 2016. URL `http://arxiv.org/abs/1602.01595`. arXiv: 1602.01595.

Mikel Artetxe, Gorka Labaka, and Eneko Agirre. Learning principled bilingual mappings of word embeddings while preserving monolingual invariance. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2289–2294, Austin, Texas, November 2016. Association for Computational Linguistics. URL `https://aclweb.org/anthology/D16-1250`.

Mikel Artetxe, Gorka Labaka, and Eneko Agirre. Learning bilingual word embeddings with (almost) no bilingual data. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 451–462, Vancouver, Canada, July 2017. Association for Computational Linguistics. URL `http://aclweb.org/anthology/P17-1042`.

Mikel Artetxe, Gorka Labaka, and Eneko Agirre. A robust self-learning method for fully unsupervised cross-lingual mappings of word embeddings. *arXiv:1805.06297 [cs]*, May 2018. URL `http://arxiv.org/abs/1805.06297`. arXiv: 1805.06297.

Peter Auer. *Code-Switching in Conversation: Language, Interaction and Identity*. Routledge, July 2013. ISBN 978-1-134-60673-3.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Irshad Ahmad Bhat, Riyaz Ahmad Bhat, Manish Shrivastava, and Dipti Misra Sharma. Joining Hands: Exploiting Monolingual Treebanks for Parsing of Code-mixing Data. *arXiv:1703.10772 [cs]*, March 2017.

Irshad Ahmad Bhat, Riyaz Ahmad Bhat, Manish Shrivastava, and Dipti Misra Sharma. Universal Dependency Parsing for Hindi-English Code-switching. *arXiv:1804.05868 [cs]*, April 2018.

Joachim Bingel and Anders Søgaard. Identifying beneficial task relations for multi-task learning in deep neural networks. *arXiv preprint arXiv:1702.08303*, 2017.

Johannes Bjerva, Barbara Plank, and Johan Bos. Semantic tagging with deep residual networks. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, pages 3531–3541. The COLING 2016 Organizing Committee, 2016. URL `http://www.aclweb.org/anthology/C16-1333`.

Igor Boguslavsky, Svetlana Grigorieva, Nikolai Grigoriev, Leonid Kreidlin, and Nadezhda Frid. Dependency treebank for russian: Concept, tools, types of information. In *Proceedings of the 18th conference on Computational linguistics-Volume 2*, pages 987–991. Association for Computational Linguistics, 2000.

Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017. ISSN 2307-387X.

Sabine Buchholz and Erwin Marsi. Conll-x shared task on multilingual dependency parsing. In *Proceedings of the tenth conference on computational natural language learning*, pages 149–164. Association for Computational Linguistics, 2006.

Antoinette Camilleri. Language values and identities: Code switching in secondary classrooms in malta. *Linguistics and education*, 8(1):85–103, 1996.

Özlem Çetinoğlu. A Turkish-German Code-Switching Corpus. page 6, 2016.

Özlem Çetinoğlu and ÇağrıÇöltekin. Part of Speech Annotation of a Turkish-German Code-Switching Corpus. In *Proceedings of the 10th Linguistic Annotation Workshop Held in Conjunction with ACL 2016 (LAW-X 2016)*, pages 120–130, Berlin, Germany, August 2016. Association for Computational Linguistics.

Jason PC Chiu and Eric Nichols. Named entity recognition with bidirectional lstm-cnns. *arXiv preprint arXiv:1511.08308*, 2015.

Y. J. Chu and T. H. Liu. On the shortest arborescence of a directed graph. *Science Sinica*, 14, 1965a.

Y. J. Chu and T. H. Liu. On the shortest arborescence of a directed graph. *Science Sinica*, 14:1396–1400, 1965b.

Alexis Conneau and Douwe Kiela. SentEval: An Evaluation Toolkit for Universal Sentence Representations. *arXiv:1803.05449 [cs]*, March 2018. URL `http://arxiv.org/abs/1803.05449`. arXiv: 1803.05449.

Alexis Conneau, Douwe Kiela, Holger Schwenk, Loïc Barrault, and Antoine Bordes. Supervised learning of universal sentence representations from natural language inference data. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 670–680, Copenhagen,

Denmark, September 2017a. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/D17-1070.

Alexis Conneau, Guillaume Lample, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. Word Translation Without Parallel Data. *arXiv:1710.04087 [cs]*, October 2017b. URL http://arxiv.org/abs/1710.04087. arXiv: 1710.04087.

Ryan Cotterell, Adithya Renduchintala, Naomi Saphra, and Chris Callison-Burch. An Algerian Arabic-French Code-Switched Corpus. page 4, 2014.

Marie-Catherine de Marneffe and Christopher D. Manning. The Stanford typed dependencies representation. pages 1–8. Association for Computational Linguistics, 2008. ISBN 978-1-905593-50-7. doi: 10.3115/1608858.1608859.

Anik Dey and Pascale Fung. A Hindi-English code switching corpus. page 4, 2014.

Nina Dongen. Analysis and prediction of dutch-english code-switching in dutch social media messages. 2017.

Timothy Dozat and Christopher D. Manning. Deep Biaffine Attention for Neural Dependency Parsing. *arXiv:1611.01734 [cs]*, November 2016. URL http://arxiv.org/abs/1611.01734. arXiv: 1611.01734.

Timothy Dozat, Peng Qi, and Christopher D Manning. Stanford's graph-based neural dependency parser at the conll 2017 shared task. *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 20–30, 2017.

Jack Edmonds. Optimum branchings. *Journal of Research of the national Bureau of Standards B*, 71(4):233–240, 1967.

Gibson Ferguson. Classroom code-switching in post-colonial contexts: Functions, attitudes and policies. *AILA review*, 16(1):38–51, 2003.

John Rupert Firth. Applications of general linguistics. *Transactions of the Philological Society*, 56(1):1–14, 1957.

Mikel L Forcada, Mireia Ginestí-Rosell, Jacob Nordfalk, Jim O'Regan, Sergio Ortiz-Rojas, Juan Antonio Pérez-Ortiz, Felipe Sánchez-Martínez, Gema Ramírez-Sánchez, and Francis M Tyers. Apertium: a free/open-source platform for rule-based machine translation. *Machine translation*, 25(2):127–144, 2011.

Sabrina Francesconi. Language habits, domains, competence and awareness: the role and use of english in malta. *English, But Not Quite: Locating Linguistic Diversity*, 1:257, 2010.

Yaroslav Ganin and Victor Lempitsky. Unsupervised Domain Adaptation by Backpropagation. *arXiv:1409.7495 [cs, stat]*, September 2014. URL http://arxiv.org/abs/1409.7495. arXiv: 1409.7495.

Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario Marchand, and Victor Lempitsky. Domain-Adversarial Training of Neural Networks. *arXiv:1505.07818 [cs, stat]*, May 2015. URL `http://arxiv.org/abs/1505.07818`. arXiv: 1505.07818.

Souvick Ghosh, Satanu Ghosh, and Dipankar Das. Part-of-speech Tagging of Code-Mixed Social Media Text. In *Proceedings of the Second Workshop on Computational Approaches to Code Switching*, pages 90–97, Austin, Texas, November 2016. Association for Computational Linguistics. URL `http://aclweb.org/anthology/W16-5811`.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

Edouard Grave, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomas Mikolov. Learning word vectors for 157 languages. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*, 2018.

Gualberto Guzmán, Joseph Ricard, Jacqueline Serigos, Barbara E Bullock, and Almeida Jacqueline Toribio. Metrics for modeling code-switching across corpora. 2017.

Jan Hajič, Eva Hajičová, Marie Mikulová, and Jiří Mírovskỳ. Prague dependency treebank. In *Handbook of Linguistic Annotation*, pages 555–594. Springer, 2017.

Injy Hamed, Mohamed Elmahdy, and Slim Abdennadher. Collection and Analysis of Code-switch Egyptian Arabic-English Speech Corpus. page 5, 2018.

Sepp Hochreiter. The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 06(02):107–116, April 1998. ISSN 0218-4885. doi: 10.1142/S0218488598000094. URL `https://www.worldscientific.com/doi/abs/10.1142/S0218488598000094`.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Aaron Jaech, George Mulcaire, Mari Ostendorf, and Noah A. Smith. A Neural Model for Language Identification in Code-Switched Tweets. pages 60–64. Association for Computational Linguistics, 2016. doi: 10.18653/v1/W16-5807. URL `http://aclweb.org/anthology/W16-5807`.

Yoon Kim, Yacine Jernite, David Sontag, and Alexander M Rush. Character-aware neural language models. 2016.

Ryan Kiros, Yukun Zhu, Ruslan Salakhutdinov, Richard S. Zemel, Antonio Torralba, Raquel Urtasun, and Sanja Fidler. Skip-Thought Vectors. *arXiv:1506.06726 [cs]*, June 2015. URL `http://arxiv.org/abs/1506.06726`. arXiv: 1506.06726.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

Omer Levy and Yoav Goldberg. Dependency-based word embeddings. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, volume 2, pages 302–308, 2014.

KyungTae Lim and Thierry Poibeau. A System for Multilingual Dependency Parsing based on Bidirectional LSTM Feature Representations. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 63–70, Vancouver, Canada, August 2017. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/K17-3006`.

Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. A structured self-attentive sentence embedding. *arXiv preprint arXiv:1703.03130*, 2017.

Lajanugen Logeswaran and Honglak Lee. An efficient framework for learning sentence representations. *arXiv:1803.02893 [cs]*, March 2018. URL `http://arxiv.org/abs/1803.02893`. arXiv: 1803.02893.

Marc Moreno Lopez and Jugal Kalita. Deep Learning applied to NLP. *arXiv:1703.03091 [cs]*, March 2017. URL `http://arxiv.org/abs/1703.03091`. arXiv: 1703.03091.

Dau-Cheng Lyu, Tien-Ping Tan, Eng-Siong Chng, and Haizhou Li. Mandarin–English code-switching speech corpus in South-East Asia: SEAME. *Language Resources and Evaluation*, 49(3):581–600, September 2015. ISSN 1574-020X, 1574-0218. doi: 10.1007/s10579-015-9303-x.

Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

Ryan McDonald, Joakim Nivre, Yvonne Quirmbach-Brundage, Yoav Goldberg, Dipanjan Das, Kuzman Ganchev, Keith Hall, Slav Petrov, Hao Zhang, Oscar Täckström, et al. Universal dependency annotation for multilingual parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, volume 2, pages 92–97, 2013.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

Lesley Milroy and Pieter Muysken. *One Speaker, Two Languages: Cross-Disciplinary Perspectives on Code-Switching*. Cambridge University Press, August 1995. ISBN 978-0-521-47912-7.

Ishan Misra, Abhinav Shrivastava, Abhinav Gupta, and Martial Hebert. Cross-stitch Networks for Multi-task Learning. *arXiv:1604.03539 [cs]*, April 2016. URL `http://arxiv.org/abs/1604.03539`. arXiv: 1604.03539.

Giovanni Molina, Fahad AlGhamdi, Mahmoud Ghoneim, Abdelati Hawwari, Nicolas Rey-Villamizar, Mona Diab, and Thamar Solorio. Overview for the Second Shared Task on Language Identification in Code-Switched Data. In *Proceedings of the Second Workshop on Computational Approaches to Code Switching*, pages 40–49, Austin, Texas, November 2016. Association for Computational Linguistics. URL `http://aclweb.org/anthology/W16-5805`.

Carol Myers-Scotton. Comparing codeswitching and borrowing. *Journal of Multilingual and Multicultural Development*, 13(1-2):19–39, January 1992. ISSN 0143-4632. doi: 10.1080/01434632.1992.9994481.

Kovida Nelakuditi, Divya Sai Jitta, and Radhika Mamidi. Part-of-Speech Tagging for Code Mixed English-Telugu Social Media Data. In *Computational Linguistics and Intelligent Text Processing*, Lecture Notes in Computer Science, pages 332–342. Springer, Cham, April 2016. ISBN 978-3-319-75476-5 978-3-319-75477-2. doi: 10.1007/978-3-319-75477-2_23.

Joakim Nivre. Dependency grammar and dependency parsing.

Joakim Nivre, Johan Hall, Jens Nilsson, Atanas Chanev, Gülşen Eryigit, Sandra Kübler, Svetoslav Marinov, and Erwin Marsi. Maltparser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, 13(2):95–135, 2007.

Joakim Nivre, Jan Hajic, McDonald Ryan, Christopher D Manning, Slav Petrov, Sampo Pyysalo, Natalia Silveira, Reut Tsarfaty, and Daniel Zeman. Universal Dependencies v1: A Multilingual Treebank Collection. page 8, 2016.

Evangelos Papalexakis, Dong Nguyen, and A Seza Doğruöz. Predicting code-switching in multilingual communication for immigrant communities. In *Proceedings of The First Workshop on Computational Approaches to Code Switching*, pages 42–50, 2014.

Niko Partanen, KyungTae Lim, Michael Rießler, and Thierry Poibeau. Dependency parsing of code-switching data with cross-lingual feature representations. In *Proceedings of the Fourth International Workshop on Computatinal Linguistics of Uralic Languages*, pages 1–17, 2018.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the diculty of training recurrent neural networks. page 9, 2013.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

Raj Nath Patel, Prakash B. Pimpale, and M. Sasikumar. Recurrent Neural Network based Part-of-Speech Tagger for Code-Mixed Social Media Text. *arXiv:1611.04989 [cs]*, November 2016. URL `http://arxiv.org/abs/1611.04989`. arXiv: 1611.04989.

Zhongyi Pei, Zhangjie Cao, Mingsheng Long, and Jianmin Wang. Multi-adversarial domain adaptation. 2018.

W. Keith Percival. Reflections on the History of Dependency Notions in Linguistics. *Historiographia Linguistica*, 17(1):29–47, January 1990. ISSN 0302-5160, 1569-9781. doi: 10.1075/hl.17.1-2.05per.

Slav Petrov, Dipanjan Das, and Ryan McDonald. A universal part-of-speech tagset. *arXiv preprint arXiv:1104.2086*, 2011.

Barbara Plank, Anders Søgaard, and Yoav Goldberg. Multilingual part-of-speech tagging with bidirectional long short-term memory models and auxiliary loss. *arXiv preprint arXiv:1604.05529*, 2016.

Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

Sebastian Ruder. An Overview of Multi-Task Learning in Deep Neural Networks. *arXiv:1706.05098 [cs, stat]*, June 2017. URL `http://arxiv.org/abs/1706.05098`. arXiv: 1706.05098.

Sebastian Ruder and Barbara Plank. Strong Baselines for Neural Semi-supervised Learning under Domain Shift. *arXiv:1804.09530 [cs, stat]*, April 2018. URL `http://arxiv.org/abs/1804.09530`. arXiv: 1804.09530.

Sebastian Ruder, Joachim Bingel, Isabelle Augenstein, and Anders Søgaard. Learning what to share between loosely related tasks. *arXiv:1705.08142 [cs, stat]*, May 2017a. URL `http://arxiv.org/abs/1705.08142`. arXiv: 1705.08142.

Sebastian Ruder, Ivan Vulić, and Anders Søgaard. A Survey Of Cross-lingual Word Embedding Models. *arXiv:1706.04902 [cs]*, June 2017b. URL `http://arxiv.org/abs/1706.04902`. arXiv: 1706.04902.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

Pingali Sailaja. Hinglish: code-switching in indian english. *ELT journal*, 65(4): 473–480, 2011.

Younes Samih, Suraj Maharjan, Mohammed Attia, Laura Kallmeyer, and Thamar Solorio. Multilingual Code-switching Identification via LSTM Recurrent Neural Networks, 2016. URL `https://ai.google/research/pubs/pub45676`.

Claude Elwood Shannon. A mathematical theory of communication. *ACM SIGMOBILE mobile computing and communications review*, 5(1):3–55, 2001.

Rouzbeh Shirvani, Mario Piergallini, Gauri Shankar Gautam, and Mohamed Chouikha. The Howard University System Submission for the Shared Task in Language Identification in Spanish-English Codeswitching. In *Proceedings of the Second Workshop on Computational Approaches to Code Switching*, pages 116–120, Austin, Texas, November 2016. Association for Computational Linguistics. URL `http://aclweb.org/anthology/W16-5815`.

Aung Si. A diachronic investigation of hindi–english code-switching, using bollywood film scripts. *International Journal of Bilingualism*, 15(4):388–407, 2011.

Anders Søgaard. What i think when i think about treebanks. In *Proceedings of the 16th International Workshop on Treebanks and Linguistic Theories*, pages 161–166, 2017. URL `http://aclweb.org/anthology/W17-7620`.

Thamar Solorio and Yang Liu. Learning to predict code-switching points. page 973. Association for Computational Linguistics, 2008a. doi: 10.3115/1613715.1613841. URL `http://portal.acm.org/citation.cfm?doid=1613715.1613841`.

Thamar Solorio and Yang Liu. Part-of-speech tagging for English-Spanish code-switched text. page 1051. Association for Computational Linguistics, 2008b. doi: 10.3115/1613715.1613852.

Anders Søgaard, Sebastian Ruder, and Ivan Vulić. On the Limitations of Unsupervised Bilingual Dictionary Induction. *arXiv:1805.03620 [cs, stat]*, May 2018. URL `http://arxiv.org/abs/1805.03620`. arXiv: 1805.03620.

Soroush Vosoughi, Prashanth Vijayaraghavan, and Deb Roy. Tweet2vec: Learning tweet embeddings using character-level cnn-lstm encoder-decoder. In *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*, pages 1041–1044. ACM, 2016.

Yogarshi Vyas, Spandana Gella, Jatin Sharma, Kalika Bali, and Monojit Choudhury. POS Tagging of English-Hindi Code-Mixed Social Media Content. pages 974–979. Association for Computational Linguistics, 2014. doi: 10.3115/v1/D14-1105.

Michael Wick, Pallika Kanani, and Adam Craig Pocock. Minimally-constrained multilingual embeddings via artificial code-switching. 2015.

Daniel Zeman. Reusable tagset conversion using tagset drivers. In *LREC*, volume 2008, pages 28–30, 2008.

Daniel Zeman, Martin Popel, Milan Straka, Jan Hajic, Joakim Nivre, Filip Ginter, Juhani Luotolahti, Sampo Pyysalo, Slav Petrov, Martin Potthast, Francis Tyers, Elena Badmaeva, Memduh Gokirmak, Anna Nedoluzhko, Silvie Cinkova, Jan Hajic jr., Jaroslava Hlavacova, Václava Kettnerová, Zdenka Uresova, Jenna Kanerva, Stina Ojala, Anna Missilä, Christopher D. Manning, Sebastian Schuster, Siva Reddy, Dima Taji, Nizar Habash, Herman Leung, Marie-Catherine de Marneffe, Manuela Sanguinetti, Maria Simi, Hiroshi Kanayama, Valeria dePaiva, Kira Droganova, Héctor Martínez Alonso, Çağrı Çöltekin, Umut Sulubacak, 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 Strnadová, Esha Banerjee, Ruli Manurung, Antonio Stella, Atsuko Shimada, Sookyoung Kwak, Gustavo Mendonca, Tatiana Lando, Rattima Nitisaroj, and Josie Li. CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal

Dependencies. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 1–19, Vancouver, Canada, August 2017. Association for Computational Linguistics.

Lidan Zhang and Kwok Ping Chan. Dependency Parsing with Energy-based Reinforcement Learning. In *Proceedings of the 11th International Conference on Parsing Technologies (IWPT'09)*, pages 234–237, Paris, France, October 2009. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W09-3838`.

Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *Advances in neural information processing systems*, pages 649–657, 2015.

Zhisong Zhang, Hai Zhao, and Lianhui Qin. Probabilistic Graph-based Dependency Parsing with Convolutional Neural Network. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1382–1392, Berlin, Germany, August 2016. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P16-1131`.

Anna V. Zhiganova. The Study of the Perception of Code-switching to English in German Advertising. *Procedia - Social and Behavioral Sciences*, 236:225–229, December 2016. ISSN 1877-0428. doi: 10.1016/j.sbspro.2016.12.011.

List of Figures

1.1 Universal dependency relations as of v2.0; source `universaldependencies.org/u/dep/` . . . 7

2.1 A basic example of a neuron with four inputs . . . 15
2.2 A neural network with one hidden layer and two inputs . . . 16
2.3 Generation of a dense (size 3) embeddings matrix from a four-word sentence, with a vocabulary size of 5. . . . 18
2.4 CBOW vs. skip-gram models for a vocabulary of size 5, context of size 3 and embedding dimension of size 3 . . . 20
2.5 Simple RNN block with three RNN cells; h_0 is the initial hidden state which can either be learnt or initialised randomly . . . 21
2.6 Architecture of an LSTM cell. Flow is indicated with arrows; lines are labelled appropriately with intersecting text . . . 22
2.7 A composition of characters for the word 'like' being fed upstream to our parser (which is described better in Figure 2.10) . . . 24
2.8 Hard sharing parameters for two tasks . . . 25
2.9 Soft parameter sharing for the same tasks . . . 25
2.10 Architecture of our dependency parser; colours indicate specific tokens . . . 26

3.1 Conneau et al.'s [2017b] approach to adversarially learning embedding mappings . . . 35
3.2 Two variable methods to supply language information: dense embeddings and language IDs, for the sentence '*this is a grapefruit*' . 44
3.3 Architectural block diagram of our MTL system; the output of our embeddings layer propagates both to our parser and to a multilayer perceptron that attempts to predict language . . . 49
3.4 Domain prediction system; arrows indicate direction of backpropagation . . . 50
3.5 Learning from development data in three steps . . . 54

4.1 Architectures for supplying no language information, and unidirectional LSTMs . . . 61
4.2 Uni+biLSTMs, and biLSTMs + embeddings . . . 62

List of Tables

2.1 Hyperparameters for our parser; ones that differ from the original are indicated with asterisks 29

3.1 Basic analytical statistics for our treebanks (combined dev and test splits 32
3.2 Code-switching statistics for our treebanks; note that Komi/Russian have only a single unified, normalised test treebank 33
3.3 Baselines, with and without character representations included . . 34
3.4 Results for parsing Catalan with various levels of training data augmentation 37
3.5 Results for parsing Catalan without other mapped training data . 37
3.6 (Compressed) embedding performance 38
3.7 Comparable uncompressed embedding performance 39
3.8 Deprel frequency per treebank 40
3.9 Deprel frequency, decomposed by arc direction 41
3.10 F_1 scores for our stochastically generated scrambling system . . . 43
3.11 F_1 scores for Hindi/English 45
3.12 F_1 scores for Komi/Russian 46
3.13 Sharing semantic tagging with dependency parsing; 'dual' LSTMs refer to LSTMs that have both shared and unshared components . 47
3.14 Hyperparameters for the language prediction section of our network 48
3.15 F_1 scores for language IDs supplied as embeddings 49
3.16 Baselines for standard parsing with additional dev data 51
3.17 Results for our domain shift system 52
3.18 Development learning results for different values of λ 56

4.1 Naive corpus statistics for both our training corpora 58
4.2 Code-switching statistics for our treebanks; note that Komi/Russian have only a single unified, normalised test treebank 59
4.3 Code-switching statistics for our treebanks; note that Komi/Russian have only a single unified, normalised test treebank 59
4.4 Hyperparameters for our code-switch predictor 60
4.5 Results for our four systems for code-switch prediction 60
4.6 Brief descriptions per experiment for the configuration with the best relevant result 63
4.7 Competitive results; Komi results are the weighted average of the two splits presented in the original paper 63
4.8 Results for parsing Hindi/English when trained on an actual training split 65