Exploring Recurrent Neural Network Grammars for Parsing Low-Resource Languages

Kemal Maulana Kurniawan

Master of Science
Artificial Intelligence
School of Informatics
University of Edinburgh

2017

Abstract

Parsing is an important task in natural language processing. A recently proposed neural parser for constituency parsing is recurrent neural network grammar (RNNG). RNNGs proved to perform very well for English, beating state-of-the-art results for supervised constituency parsing. However, for other languages, RNNGs' effectiveness has yet to be discovered. Unlike English, most languages are low-resource. This thesis describes our research in exploring RNNGs to parse low-resource languages, which is represented by Indonesian. Working with a low-resource language means dealing with sparsity problem. We addressed the issue by employing character embeddings. Our results show that character embeddings make RNNGs faster to train, but at the expense of a small performance drop. Also, we found that RNNGs work surprisingly well in low-data scenario, disputing the preconception that neural-based models need a lot of training data to function well.

Acknowledgements

Bismillahirrahmanirrahim.

First and foremost, I wholeheartedly thank my supervisor, Frank Keller, for the invaluable guidance and support since the very beginning to the very end of this work. Frank has been very accessible; he always responded to my inquiries. He advised me, kept me on track, and helped me figure out the way out when I was stuck. He was also willing to answer questions not pertinent to the work. Without his support, I would never be able to proudly complete this thesis. And for that, I will forever be indebted to him.

I thank my parents, who always believe in, pray for, and support me.

I also thank my fellow informatics students, with whom I spent the past year. Special thanks go to Pramudya Ananto and Muhammad Arrasy Rahman, for the warm company and stimulating discussions.

I am very grateful to all my friends, especially my badminton squad. We spent so many times playing badminton, making my days here in Edinburgh more happy and also healthy.

I am also grateful to: Coldplay, for holding a fantastic concert in Cardiff to which I attended; Fine Bros Entertainment, for making an entertaining reaction video every day; and HBO, for the best show ever, Game of Thrones, which suceeded in making every week of dissertation writing exciting, despite the approaching deadline.

This work was possible due to the generous funding from Lembaga Pengelola Dana Pendidikan (LPDP), Ministry of Finance, Republic of Indonesia.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Kemal Maulana Kurniawan*)

To the giants, whose shoulders I am standing on.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis describes an exploration of recurrent neural network grammars for constituency parsing. We explored how well recurrent neural network grammars parse non-English languages, especially low-resource ones. We chose Indonesian as a sample of such a language. Also, we employed a character embedding model to mitigate the sparsity problem in such a low-data scenario.

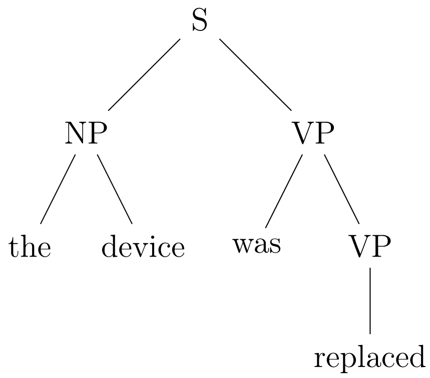
In this chapter, we present the motivation for this research and discuss the objectives we seek to achieve. It is then closed by a brief overview of the thesis.

1.1 Motivation

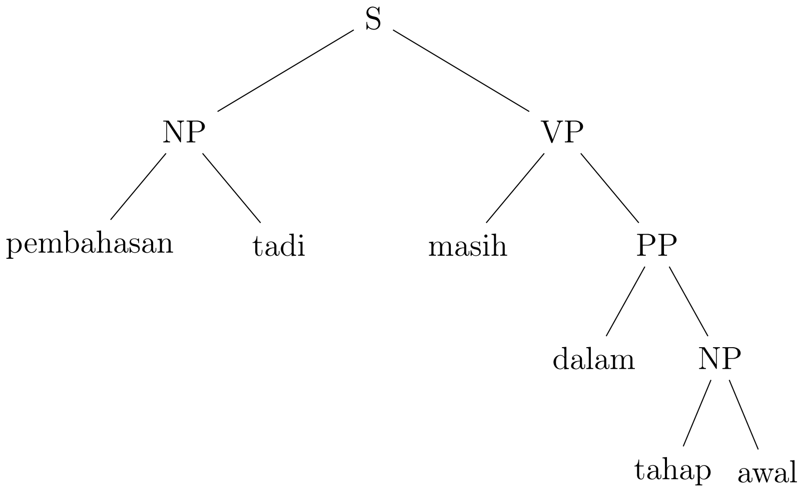
Parsing is an important task in natural language processing. It aims to assign a structure, usually a syntax tree, to a sentence. For example, an English sentence *the device was replaced* might be assigned a syntax tree declaring that *the device* is a noun phrase, *was replaced* is a verb phrase, and their concatenation is a sentence. An Indonesian sentence *pembahasan tadi masih dalam tahap awal*[1] might be treated similarly: *pembahasan tadi* is a noun phrase, *masih dalam tahap awal* is a verb phrase, and they join to form a sentence. Parse trees of the two sentences are shown in Figure 1.1. Such parse trees are generally useful as features for downstream tasks such as semantic role labeling, which seeks to discover semantic roles in a sentence (Gildea and Jurafsky, 2002). Syntactic information

[1]English: *the previous discussion is still in the early stage.*

is found to be very useful for identifying semantic arguments (Punyakanok et al., 2008). Therefore, parsing is an essential problem in natural language processing.



(a) A syntax tree of the English sentence *the device was replaced.*



(b) A syntax tree of the Indonesian sentence *pembahasan tadi masih dalam tahap awal.*

Figure 1.1: Sample syntax trees of an English and an Indonesian sentence.

Recurrent neural network grammars (RNNGs) (Dyer et al., 2016) recently achieved state-of-the-art result in English constituency parsing. Constituency parsing attempts to assign a sentence to a syntax tree describing the constituents in the sentence and how they are combined. RNNGs are essentially shift-reduce parsers, but they ingeniously use neural networks to select the correct action in each step of the parsing algorithm. RNNGs use vector embeddings to represent words, nonterminals, and incomplete syntax trees. Thus, RNNGs suffer from out-of-vocabulary problem: when there are unseen words at test time. A typical solution is to replace rare words in the training set with special tokens for unknown words, learn the embeddings of the tokens at training time, and treat unseen words at test time as these special tokens. However, this approach is completely oblivious to the similarity between the seen and unseen words because their embeddings are learned separately. A better approach is to exploit this similarity. For instance, the word *business* is very similar to the word *businesses*. So, if we only have *business* in the training set, we would like the parser to infer the embedding of *businesses* more accurately by noticing that the two words are morphologically related.

While RNNGs have been proven to perform well in parsing English, its effectiveness in parsing other languages has yet to be discovered. Unlike English, most languages in the world are low-resource, i.e. there are only few (sometimes none at all) linguistic resources (e.g. annotated corpora) that can be employed in natural language processing tasks. Therefore, we are interested in seeing how well RNNGs work on low-resource languages. One such language is Indonesian, which we used in this work. We chose Indonesian because of its low number of resources for parsing and our familiarity with the language.

One major problem in dealing with low-resource languages is sparsity. Out-of-vocabulary problem is one instance of sparsity problem. Sparsity is already a serious problem for high-resource languages such as English, let alone low-resource ones. We addressed this problem by incorporating character embeddings. Several studies have demonstrated that character embeddings are sufficient in many language tasks (dos Santos and Zadrozny, 2014; Ling et al., 2015; Ballesteros et al., 2015; Kim et al., 2016; Vylomova et al., 2016; Lee et al., 2016). In dependency parsing using shift-reduce algorithm, for instance, character embeddings outperform word embeddings in many languages (Ballesteros et al., 2015). In these tasks, a word embedding is computed by composing the embeddings of its characters. Assuming that the training set contains all the characters in the language, unseen words at test time do not need special treatment; their embeddings can be computed in the same way as any seen words. Thus, out-of-vocabulary problem naturally vanishes. Additionally, the number of characters is usually much lower than the number of words. Hence, the storage space required to store the embeddings is much more efficient. On top of that, the sparsity problem is severely reduced because now we have much fewer parameters to estimate. Likewise, the previous assumption that the training set contains all the characters are also quite frequently satisfied in practice, precisely because working with character embeddings makes the training data less sparse.

1.2 Objectives

The objectives of this research are:

- Investigating how RNNGs perform in both low-data and non-English language scenario, with Indonesian as a sample of such a language.
- Exploring the use of character embeddings for RNNGs by implementing a character embedding model and comparing it to the original RNNGs that use word embeddings.

1.3 Overview

This thesis describes our research that is motivated by reasons presented in Section 1.1. We seek to achieve our objectives that has been mentioned in Section 1.2.

In Chapter 2, we provide the necessary background to understand our approaches. Specifically, Section 2.1 explains about constituency parsing, what it is and how it differs from dependency parsing. Section 2.2 describes recurrent neural networks, including long short-term memory networks, which are the main neural network models used in RNNGs. The subsequent section, Section 2.3, describes recurrent neural network grammars in more detail. Section 2.4 presents a character embedding model that we used in this research.

In Chapter 3, the methodology used in this research is presented. Starting from Section 3.1, we mention the baseline models in this work. Section 3.2 explains about the evaluation metric used to assess our models. Section 3.3 lists several datasets that we used including the preprocessing steps applied. In the subsequent section, Section 3.4, we describe our implementation details. The section is then closed by a description of our experiments setup in Section 3.5.

Chapter 4 is where we present our results and discussions. The chapter is started by a presentation of our results and closed by discussions on them. The chapter afterward, Chapter 5, is the last chapter of this thesis. That chapter is where we conclude our work and state possible directions for future work.

Chapter 2

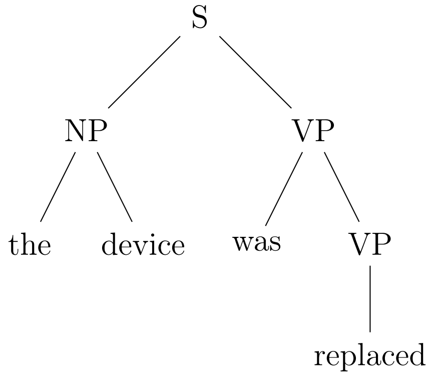
Background

In this chapter, we describe what constituency parsing is and how it differs from another kind of parsing, namely dependency parsing. This description is then followed by an explanation on recurrent neural networks, and its variant, long short-term memory networks. Next, we are ready to cover recurrent neural network grammars, both the discriminative and the generative variants. Lastly, we close this chapter with an explanation on character embeddings and one example of character embedding model we used in this research.

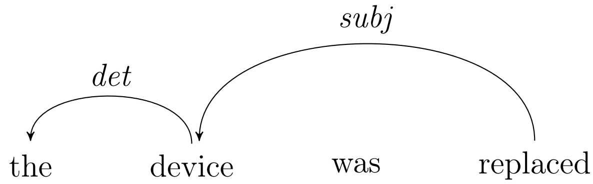
2.1 Constituency vs dependency parsing

In our parsing example in Chapter 1, we said that *the device* is a noun phrase (NP), *was replaced* is a verb phrase (VP), and together they constitute a sentence (S). This example illustrates a possible output of a constituency parser. Thus, constituency parsing can be defined loosely as a kind of parsing whose purpose is to discover these constituents and the way they are joined in the sentence. This information is usually represented as a syntax tree. Figure 2.1a shows an example of such a tree.

A constituent is described as a group of words that can be considered a single unit, and constituents having the same label (e.g. NP) can occur in similar syntactic contexts (Jurafsky and Martin, 2009). For instance, the noun phrase in the example above can be substituted by another noun phrase *the wooden table*, producing a valid sentence *the wooden table was replaced*. This example shows



(a) Syntax tree



(b) Dependency tree

Figure 2.1: Example of the different parse trees for the sentence *the device was replaced.*

a characteristic of constituency parsing: it is only applicable to languages whose grammar is able to express ordering rules over constituents. In this case, one such rule might be *a noun phrase can be followed by a verb phrase to form a sentence.*

Dependency parsing, on the other hand, does not rely on the grammar being able to express ordering. It only cares about the binary relation between the words in the sentence (Jurafsky and Martin, 2009). Dependency parsers tie two words in the sentence to form a relation and label the said relation. These relations are usually asymmetric, and when drawn in a tree, they are represented as directed arcs from the head word to the dependent word. Figure 2.1b shows an example of a dependency tree.

In our example sentence *the device was replaced*, as shown in Figure 2.1b, a dependency parser might link *replaced* to *device* and label the relation as *subj*, meaning that *device* is the subject of the verb *replaced*. The parser might also link *device* to *the* and label the relation as *det*, meaning that *the* is the determiner of the noun *device*. From this illustration, we can see how ordering does not matter as much in dependency parsing. All the dependency parser does is linking words, regardless of their positions. As a result, dependency parsing makes more sense

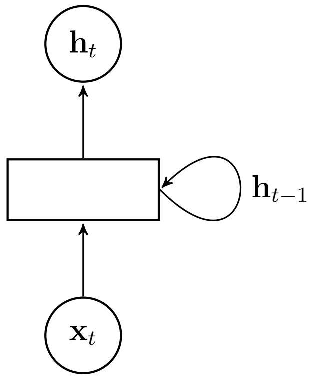


Figure 2.2: Architecture of an RNN. To compute the hidden state $\mathbf{h}_t$, the previous hidden state $\mathbf{h}_{t-1}$ is included as input along with $\mathbf{x}_t$.

for languages that have relatively free word ordering, such as Czech and Japanese.

2.2 Recurrent neural networks

Recurrent neural networks (RNNs) are neural networks designed to model dependency in an input sequence (Elman, 1990). They are characterized by their recurrent connection which enables hidden state from the previous time step to be included as input to the current time step. This inclusion allows information to flow from one time step to the next, which gives RNNs their ability to model dependency in the sequence. An illustration of an RNN architecture is shown in Figure 2.2. The figure shows a recurrent connection in an RNN, depicted as a loop, which represents how the hidden state is fed back as input to the network.

Although it does not seem so on the surface, RNNs are really not that different from regular feed-forward neural networks. The recurrent connection can be unrolled to produce a similar architecture to that of a feed-forward network, as shown in Figure 2.3. The figure shows an unrolled RNN architecture, which looks surprisingly similar to that of a 3-layer feed-forward neural network. It is not hard to imagine that unrolling an RNN for longer time steps would result in an architecture similar to that of a deep feed-forward network.

Mathematically, an RNN computation can be formulated as follows: given an input sequence $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T \in \mathbb{R}^m$, at time step t, an RNN with n hidden units

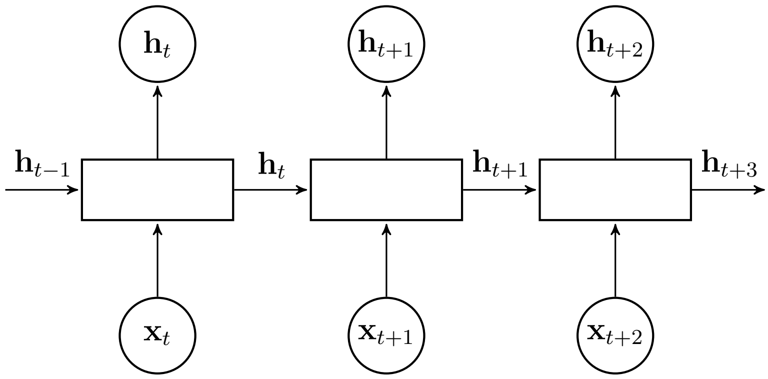


Figure 2.3: Architecture of an unrolled RNN for three time steps. This architecture is similar to that of a 3-layer feed-forward neural network.

computes (Cooijmans et al., 2016)

$$\mathbf{h}_t = \phi(\mathbf{W}_x \mathbf{x}_t + \mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{b}) \tag{2.1}$$

where $\mathbf{W}_x \in \mathbb{R}^{n \times m}, \mathbf{W}_h \in \mathbb{R}^{n \times n}, \mathbf{b} \in \mathbb{R}^n$ are parameters, and $\phi(\cdot)$ is a nonlinear activation function. The initial hidden state $\mathbf{h}_0 \in \mathbb{R}^n$ is usually initialized to a zero vector.

RNNs are usually trained by gradient-based methods such as stochastic gradient descent. Using such methods, it is necessary to compute the gradients of the parameters, either via automatic differentiation provided by many deep learning libraries or via manual derivation of their mathematical expressions. These gradient computations involve a multiplication chain, which grows longer as the time step increases. Since each term's magnitude in the multiplication chain is typically no larger than one, the gradients become minuscule when the input sequence is sufficiently long. These tiny gradients hamper the networks' ability to learn. This problem is known as the *vanishing gradient problem*, which impedes RNNs' ability to model long-term dependency in the input sequence (Hochreiter, 1991; Bengio et al., 1994).

2.2.1 Long short-term memory networks

Long short-term memory networks (LSTMs) are specially designed RNNs to combat the vanishing gradient problem (Hochreiter and Schmidhuber, 1997). LSTMs overcome the said problem by employing a *memory cell*. LSTMs are able to read from or write to this cell, regulated by several gates. These gates determine how much contribution the memory cell from the previous time step (*forget gate*) and the current input (*input gate*) have to the current memory cell value, and how much of that value is going to be the output at the current time step (*output gate*).

Formally, an LSTM computation can be formulated as follows: given a sequential input $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T \in \mathbb{R}^m$, at each time step t, an LSTM with n hidden units computes (Cooijmans et al., 2016)

$$\begin{pmatrix} \mathbf{i} \\ \mathbf{f} \\ \mathbf{o} \\ \mathbf{g} \end{pmatrix} = \mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t + \mathbf{b} \tag{2.2}$$

$$\mathbf{c}_t = \sigma(\mathbf{f}) \odot \mathbf{c}_{t-1} + \sigma(\mathbf{i}) \odot \tanh(\mathbf{g}) \tag{2.3}$$

$$\mathbf{h}_t = \sigma(\mathbf{o}) \odot \tanh(\mathbf{c}_t) \tag{2.4}$$

where $\mathbf{c}_t$ and $\mathbf{h}_t$ are the memory cell and output at time step t, $\mathbf{W}_h \in \mathbb{R}^{4n \times n}$, $\mathbf{W}_x \in \mathbb{R}^{4n \times m}$, $\mathbf{b} \in \mathbb{R}^{4n}$ are parameters, symbol $\odot$ denotes an element-wise multiplication, and function $\sigma(\cdot)$ and $\tanh(\cdot)$ are applied element-wise. $\sigma(\mathbf{f})$, $\sigma(\mathbf{i})$, and $\sigma(\mathbf{o})$ are the forget, input, and output gate respectively.

Looking at the equations closely, we see how by modulating the three gates, LSTMs are able to control how much contribution the previous or current time step has. One thing worth noticing is that the memory cell update expression in equation 2.3 is almost linear. This linearity enables the gradients to flow more easily at training time, resulting in the ability of LSTMs to mitigate the vanishing gradient problem. This ability allows LSTMs to model long-term dependency much more effectively than RNNs.

2.3 Recurrent neural network grammars

Recurrent neural network grammars (RNNGs), by Dyer et al. (2016), are constituency parsers which employ shift-reduce parsing algorithm. They use recurrent neural networks—specifically long short-term memory networks—in deciding which action to take in each step of the algorithm. Formally, an RNNG is a 3-tuple (Σ, N, Θ) where Σ is a finite set of terminal symbols, N is a finite set of nonterminal symbols (which is mutually exclusive with Σ) and Θ is a collection of neural network parameters (Dyer et al., 2016). While parsers commonly store some grammar explicitly, either provided as input or learned at training time, RNNGs' grammar is completely and implicitly specified by the parameters Θ. This parameterization of the grammar also relaxes the assumption of independence typically found in probabilistic context-free grammar parsers.

There are two variants of RNNGs. The discriminative ones attempt to directly model the probability distribution over parse trees given an input string. On the other hand, the generative ones model the joint probability of a parse tree and an input string. The next two sections explain the two variants in more detail.

2.3.1 Discriminative variants

The discriminative variants of RNNGs model the conditional probability $\Pr(y|x)$ where x and y denote an input string and a parse tree respectively. Assume that x is read from left to right. Discriminative RNNGs use two data structures: a stack S and an input buffer B. The parsing algorithm begins with an empty stack and an input buffer containing all the words of x from left to right. At each iteration, one of the following actions is selected:

- NT(X), which introduces a new incomplete nonterminal X by pushing it onto S. The top of the stack S is now X.
- SHIFT, which shifts the input buffer to the left, effectively removing the leftmost word in B, and then pushes it onto S. The top of the stack S is now the shifted word.
- REDUCE, which completes the topmost incomplete nonterminal in S by popping elements from the top of S until an incomplete nonterminal is encountered, removing the said nonterminal, and using it as the label of a new constituent which has the popped elements as its children. The new constituent is then pushed back onto S. The top of the stack S is now the said constituent, which is also a completed nonterminal.

The algorithm terminates when B is empty and S contains only one completed element. The element is the parse tree of x. Table 2.1 demonstrates an example run of the algorithm.

	Stack (S)	Input Buffer (B)	Action
1			NT(S)
2	(S	*the hungry cat meows*	NT(NP)
3	(S | (NP	*the hungry cat meows*	SHIFT
4	(S | (NP | *the*	*hungry cat meows*	SHIFT
5	(S | (NP | *the* | *hungry*	*cat meows*	SHIFT
6	(S | (NP | *the* | *hungry* | *cat*	*meows*	REDUCE
7	(S | (NP *the hungry cat*)	*meows*	NT(VP)
8	(S | (NP *the hungry cat*) | (VP	*meows*	SHIFT
9	(S | (NP *the hungry cat*) | (VP | *meows*		REDUCE
10	(S | (NP *the hungry cat*) | (VP *meows*)		REDUCE
11	(S (NP *the hungry cat*) (VP *meows*))		

Table 2.1: An example run of the discriminative RNNG parsing algorithm for the input string *the hungry cat meows*. Parse trees are written in bracket notation. Vertical bars (|) separate the stack elements (Dyer et al., 2016).

To decide what action a_t to take at iteration t, discriminative RNNGs compute a probability distribution over valid actions, conditioned on the action history $\mathbf{a}_{<t}$. They encode the parser state (the stack, buffer, and action history) into a vector embedding $\mathbf{u}_t$, and pass this vector through a softmax layer to get the distribution. Formally, this computation can be expressed as (Dyer et al., 2016)

$$\Pr(a_t|\mathbf{a}_{<t}) = \frac{\exp\left(\mathbf{r}_{a_t}^\top \mathbf{u}_t + b_{a_t}\right)}{\sum_{a' \in \mathcal{A}_D(S_t,B_t)} \exp\left(\mathbf{r}_{a'}^\top \mathbf{u}_t + b_{a'}\right)} \tag{2.5}$$

where $\mathbf{r}_a$ and b_a are the embedding and bias term of action a respectively, $\mathcal{A}_D(S,B)$ denotes the set of possible (discriminative) parser actions given stack S and input buffer B, and S_t and B_t denote the stack and input buffer state at time step t respectively. The action embeddings $\mathbf{r}_a$ and biases b_a are included as parameters in Θ. The set of possible (discriminative) parser actions is determined by the content of stack S and buffer B, whose rules are mentioned in more details in (Dyer et al., 2016). The parser state embedding vector $\mathbf{u}_t$ is defined as (Dyer et al., 2016)

$$\mathbf{u}_t = \tanh\left(\mathbf{W}\left[\mathbf{s}_t; \mathbf{o}_t; \mathbf{h}_t\right] + \mathbf{c}\right) \tag{2.6}$$

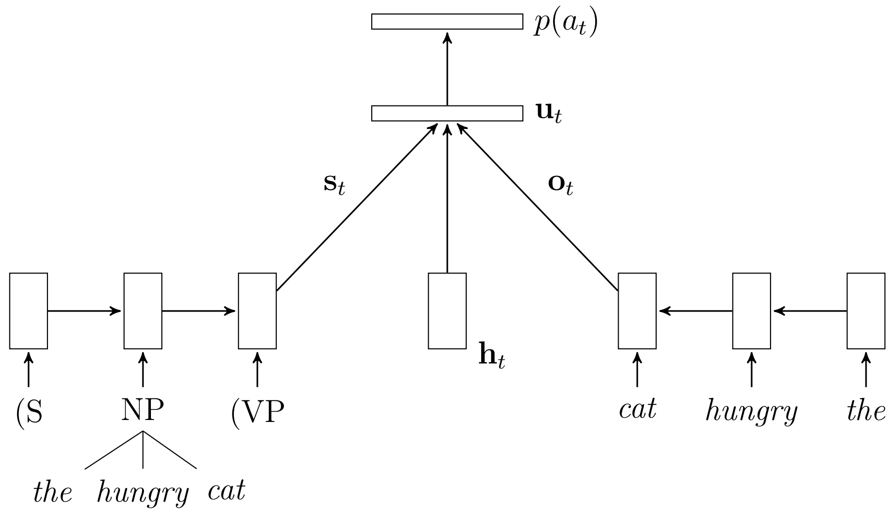


Figure 2.4: Illustration of the generative RNNG architecture to compute probability distribution over possible actions given the vector representation of the stack ($\mathbf{s}_t$), output buffer ($\mathbf{o}_t$), and action history ($\mathbf{h}_t$). This parser state corresponds to the state at line 8 in Table 2.2 (Dyer et al., 2016).

where $\mathbf{W}$ and $\mathbf{c}$ are weight matrix and bias vector respectively and included as model parameters in Θ, symbols $\mathbf{s}_t, \mathbf{o}_t, \mathbf{h}_t$ denote the embedding of S_t, B_t, and $\mathbf{a}_{<t}$ respectively. Figure 2.4 illustrates the architecture of an RNNG (it shows the generative one, which is very similar and will be explained in the next section).

The embedding $\mathbf{o}_t$ is obtained by feeding the embeddings of the words in B_t as inputs to an LSTM. Similarly, the embedding $\mathbf{h}_t$ is obtained by feeding the embeddings of the selected actions $\mathbf{a}_{<t}$ as inputs to an LSTM chronologically. Note that these action embeddings are different from the action embeddings $\mathbf{r}_a$ used in equation 2.5 (so, we have two separate embeddings for an action). Obtaining the embedding $\mathbf{s}_t$ is done rather differently, because the content of the stack can be a terminal symbol, an incomplete nonterminal, or a complete constituent in the form of a tree. Simple embeddings lookup tables can be used for the first two, but the tree embedding is obtained via a syntactic composition function that is crucial to RNNGs' performance (Kuncoro et al., 2017). This function accepts the tree's children embeddings as inputs and return the tree embedding as output, as illustrated in Figure 2.5. This composition function uses two LSTMs: forward and backward LSTMs. The forward LSTM is given the embedding of the nonterminal label as the first input, and then the embeddings of the children from left to right. The backward LSTM is also given the same set of inputs but in reversed

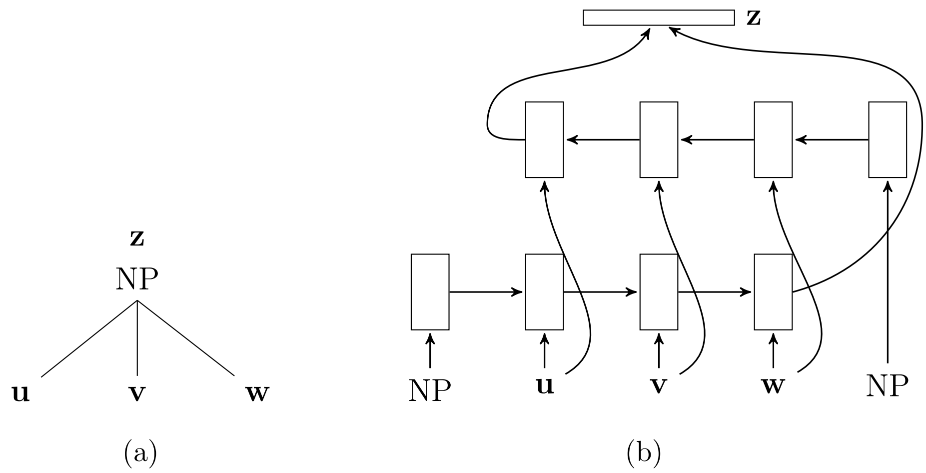


Figure 2.5: Illustration of the RNNGs' syntactic composition function. Given a tree with its children embeddings $\mathbf{u}, \mathbf{v}, \mathbf{w}$, the embedding $\mathbf{z}$ of the tree is computed by a bidirectional LSTM (Dyer et al., 2016).

order. Outputs from the two LSTMs are then concatenated and passed through an affine transformation and a tanh activation function to yield the final tree embedding. Note that because this composition requires the stack's elements to be pushed and popped, a stack LSTM (Dyer et al., 2015) is employed to be able to do those operations efficiently. The parameters of all the LSTMs, together with the weight matrix and bias term used in the affine transformation of the composition, and all the embeddings described above are included as parameters in Θ.

2.3.2 Generative variants

The generative variants of RNNGs model the joint probability of the parse tree y and the input string x. They are largely the same as the discriminative ones. The same shift-reduce parsing algorithm is used, but with two alterations:

1. There is no input buffer B; an output buffer O is used instead. The algorithm begins with an empty output buffer.
2. Instead of a SHIFT action, they have GEN(w) actions for every possible terminal symbol w. When one of these actions is selected, they generate the terminal symbol w and push it onto the stack S and the output buffer O.

<table>
<tr><th></th><th>Stack (S)</th><th>Output Buffer (O)</th><th>Action</th></tr>
<tr><td>1</td><td></td><td></td><td>NT(S)</td></tr>
<tr><td>2</td><td>(S</td><td></td><td>NT(NP)</td></tr>
<tr><td>3</td><td>(S | (NP</td><td></td><td>GEN(the)</td></tr>
<tr><td>4</td><td>(S | (NP | the</td><td>the</td><td>GEN(hungry)</td></tr>
<tr><td>5</td><td>(S | (NP | the | hungry</td><td>the hungry</td><td>GEN(cat)</td></tr>
<tr><td>6</td><td>(S | (NP | the | hungry | cat</td><td>the hungry cat</td><td>REDUCE</td></tr>
<tr><td>7</td><td>(S | (NP the hungry cat)</td><td>the hungry cat</td><td>NT(VP)</td></tr>
<tr><td>8</td><td>(S | (NP the hungry cat) | (VP</td><td>the hungry cat</td><td>GEN(meows)</td></tr>
<tr><td>9</td><td>(S | (NP the hungry cat) | (VP | meows</td><td>the hungry cat meows</td><td>REDUCE</td></tr>
<tr><td>10</td><td>(S | (NP the hungry cat) | (VP meows)</td><td>the hungry cat meows</td><td>REDUCE</td></tr>
<tr><td>11</td><td>(S (NP the hungry cat) (VP meows))</td><td>the hungry cat meows</td><td></td></tr>
</table>

Table 2.2: An example of a parse tree and sentence generation of the generative RNNG. Parse trees are written in bracket notation. Vertical bars (|) separate the stack elements (Dyer et al., 2016).

The algorithm ends when a single completed element remains in the stack S and O contains all the words in x. An example run of the algorithm is shown in Table 2.2.

The joint probability of the parse tree y and input string x, $\Pr(x, y)$, is defined as (Dyer et al., 2016)

$$\Pr(x, y) = \prod_{t=1}^{|\mathbf{a}(x,y)|} \Pr(a_t | \mathbf{a}_{<t}) \tag{2.7}$$

$$= \prod_{t=1}^{|\mathbf{a}(x,y)|} \frac{\exp\left(\mathbf{r}_{a_t}^\top \mathbf{u}_t + b_{a_t}\right)}{\sum_{a' \in \mathcal{A}_G(S_t, O_t)} \exp\left(\mathbf{r}_{a'}^\top \mathbf{u}_t + b_{a'}\right)} \tag{2.8}$$

where $\mathbf{a}(x, y)$ denotes the sequence of (generative) parser actions for input string x and parse tree y, symbol O_t denotes the output buffer at time step t, and $\mathcal{A}_G(S, O)$ is the set of possible (generative) parser actions given stack S and output buffer O. Other symbols mean the same as in the discriminative counterparts. Computing the parser state embedding $\mathbf{u}_t$ is also done in the same manner as the discriminative variants. The set of possible (generative) parser actions is detailed more clearly in (Dyer et al., 2016).

The number of GEN(w) actions is equal to the number of possible terminal symbols, which is typically a large number. To reduce this number to a more manageable size, Dyer et al. proposed a two-step approach in generating a terminal symbol. First, the action to generate is selected (GEN). Then, the terminal symbol is generated conditioned on the current parser state. They further reduced the complexity of word generation by using class-factored softmax (Baltescu and Blunsom, 2015; Goodman, 2001). The number of classes is set to $\sqrt{|\Sigma|}$, so that the resulting complexity is $O(\sqrt{|\Sigma|})$, instead of $O(|\Sigma|)$ had the full vocabulary size is used.

2.3.3 Training

Training both variants of RNNGs is done very similarly. The training data is an oracle file containing an ordered list of correct actions for every training sentence. The parser is trained using any gradient-based method, such as stochastic gradient descent, by maximizing the likelihood of the oracle. Although the training is done similarly, they correspond to different things. For the discriminative variants, since there is only SHIFT action, the likelihood being maximized is the conditional likelihood of the sequence of actions given the corresponding input string. For the generative variants however, the likelihood of generating the input string is already captured by the GEN(w) actions. Therefore, the likelihood being maximized is the joint likelihood of the input string and the sequence of actions.

2.3.4 Inference

Inference corresponds to finding a parse tree $\hat{y}$ which maximizes $\Pr(y|x)$, or formally

$$\hat{y} = \arg\max_{y} \Pr(y|x) \tag{2.9}$$

In the discriminative variants, this procedure is done using greedy algorithm. Starting with the initial parser state, action with the highest probability is selected and performed until the algorithm terminates. The resulting element in the stack is the parse tree of the input string.

As for the generative variants, we need to involve $\Pr(x, y)$ in the expression, so

$$\hat{y} = \arg\max_y \Pr(y|x) \tag{2.10}$$

$$= \arg\max_y \frac{\Pr(x, y)}{\Pr(x)} \tag{2.11}$$

$$= \arg\max_y \Pr(x, y) \tag{2.12}$$

Inference in the generative variants is then done using Monte Carlo method. Samples are drawn from the discriminative variant and then re-ranked by the generative counterpart. Sample with the highest probability under the generative variant is the parse tree of the input string. Sampling from the discriminative variant can be done efficiently by ancestral sampling procedure.

2.4 Character embedding models

Character embedding models have been found to be successful in many language tasks, such as part-of-speech tagging (dos Santos and Zadrozny, 2014; Ling et al., 2015), language modeling (Ling et al., 2015; Kim et al., 2016), machine translation (Vylomova et al., 2016; Lee et al., 2016), and dependency parsing (Ballesteros et al., 2015). In these tasks, the embedding $\mathbf{w}$ of a word w with n characters $c_1, c_2, \ldots, c_n$ can be expressed as

$$\mathbf{w} = f\left(\mathbf{u}_{c_1}, \mathbf{u}_{c_2}, \ldots, \mathbf{u}_{c_n}\right) \tag{2.13}$$

where $\mathbf{u}_c$ is the embedding of character c and f is a composition function. People experimented with this composition function f. The function can be a simple addition (Botha and Blunsom, 2014), convolution with pooling (dos Santos and Zadrozny, 2014), or a neural network model, for instance bidirectional LSTM (Ling et al., 2015; Ballesteros et al., 2015) or convolutional neural network (Kim et al., 2016).

In our research, we used the addition model. The addition model is inspired by the work of Botha and Blunsom (2014). They proposed that morphologically similar words should have similar word representations. To achieve this goal, they defined a mapping $\mu(\cdot)$ that maps a word to its sequence of factors, and each factor f has a corresponding factor vector embedding, $\mathbf{r}_f$. An embedding $\mathbf{w}$

of a word w is then defined as

$$\mathbf{w} = \sum_{f \in \mu(w)} \mathbf{r}_f \tag{2.14}$$

In their work, the mapping $\mu(\cdot)$ mapped a word to its surface morphemes. For instance, the word *imperfection* was mapped to [*im*, *perfect*, *ion*]. In our research, however, we mapped a word to its characters, so that the word *imperfection* was instead mapped to [*i*, *m*, *p*, *e*, *r*, *f*, *e*, *c*, *t*, *i*, *o*, *n*].

Chapter 3

Methodology

In this chapter, we discuss our research methodology. Firstly, we briefly mention several other parsers as baselines. Then, we describe the evaluation metric that we used in our research to measure parsing performance. Next, we discuss the datasets used and any preprocessing applied. We also cover how unknown words were handled and which pretrained word embeddings used in our work. This discussion will be followed by an explanation of our implementation including the character embedding model. Finally, we describe our experiments setup such as the initialization method, the optimizer used to train our parsers, and so on. This description includes the regularization method and sampling for inference in generative RNNGs.

3.1 Baselines

Our research built on the work by Dyer et al. (2016). So, naturally we picked their plain RNNGs as our baseline. Also, we selected RNNGs equipped with pretrained word embeddings as additional baseline. Finally, we included as baselines two state-of-the-art non-neural parsers for English, namely the Stanford factorized parser[1] (Klein and Manning, 2003) and the Berkeley parser[2] (Petrov et al., 2006). We trained all the above parsers from scratch on our datasets.

[1]Version 3.7.0: `https://nlp.stanford.edu/software/lex-parser.shtml`

[2]Version 1.7: `https://github.com/slavpetrov/berkeleyparser`

3.2 Evaluation

To evaluate parsing performance, we used a standard metric for parsing called `PARSEVAL` (Klavans et al., 1991). It measures the precision and recall of a hypothesis syntax tree against a gold reference syntax tree. A constituent in the hypothesis tree is labeled as *correct* if there is a constituent in the reference tree having the same starting point, ending point, and nonterminal label. The precision and recall can then be computed as follows (Jurafsky and Martin, 2009)

$$\text{precision} = \frac{\text{\# of correct constituents in the hypothesis}}{\text{\# of total constituents in the hypothesis}} \tag{3.1}$$

$$\text{recall} = \frac{\text{\# of correct constituents in the hypothesis}}{\text{\# of total constituents in the reference}} \tag{3.2}$$

Rather than reporting the precision and recall separately, a single F_1 score is often reported instead. An F_1 score is defined as

$$F_1 = \frac{2PR}{P + R} \tag{3.3}$$

where P and R denote the precision and recall value respectively. Thus, an F_1 score is the harmonic mean of precision and recall. In our experiments, we used the F_1 score as a measure of parsing performance. We used a software to compute this score, called `EVALB`[3], which is available online. This software was also used by Dyer et al. in their work. We run `EVALB` with the provided `COLLINS.prm` parameter file.

3.3 Datasets

For English, we used the Penn Treebank WSJ dataset with the common split: section 2–21 as the training set, section 24 as the development set, and section 23 as the test set. The number of sentences in each set is 39,832 sentences, 1,346 sentences, and 2,416 sentences respectively. As for Indonesian, we used the Indonesian Treebank dataset[4]. This dataset contains 1,029 parsed sentences

[3] `http://nlp.cs.nyu.edu/evalb/`

[4] `https://github.com/famrashel/idn-treebank`

	Training	Development	Testing
Fold 0	618	201	205
Fold 1	618	204	202
Fold 2	618	201	203
Fold 3	618	199	203
Fold 4	618	206	203

Table 3.1: The number of sentences in the training, development, and testing set for each cross-validation fold in PTB 1029 dataset.

from PAN Localization Indonesian corpora collection[5]. An example of a parse tree from the dataset is (the italicized words are not part of the tree but rather English translations of the word right above them):

```
(NP (NN (Kera))
          monkey
    (SBAR (SC (untuk))
                 for
          (S (NP-SBJ (*))
             (VP (VB (amankan))
                        secure
                 (NP (NN (pesta olahraga)))))))
                         festival    sport
```

As the example shows, the dataset uses the same nonterminal labels, grammatical function tags, and null elements specification as the Penn Treebank. One difference is that in the Indonesian Treebank dataset, there can be multiple words grouped under a single part-of-speech (POS) tag, such as `(NN (pesta olahraga))`. To handle this case, we joined the words and treated them as a single word, so it became `(NN pesta_olahraga)`.

There has not been any research that used this dataset as far as we know, so unlike Penn Treebank, there is not a common split yet. On top of that, the number of sentences is very low. Therefore, we employed 5-fold cross-validation to address this issue. To have a fair comparison between English and Indonesian, we also

[5] `http://www.panl10n.net/indonesia/#Linguistic_Resources_`

	Training	Development	Testing
Fold 0	620	206	206
Fold 1	620	206	206
Fold 2	621	205	206
Fold 3	621	205	206
Fold 4	619	203	206

Table 3.2: The number of sentences in the training, development, and testing set for each cross-validation fold in ITB full dataset.

created a small Penn Treebank dataset. We picked the first 1,029 sentences from the full Penn Treebank training set as this small dataset, and applied the same 5-fold cross-validation procedure as the Indonesian dataset. By matching the dataset size, we are sure that any difference in performance between them is likely because of the languages themselves, rather than the size of the data. The full Penn Treebank dataset, the small one, and the Indonesian Treebank dataset will be called PTB full, PTB 1029, and ITB full henceforth. Table 3.1 and 3.2 show the statistics of PTB 1029 and ITB full datasets respectively. The difference in total number of sentences for each fold is because we removed sentences from development and testing set which have nonterminal labels or `UNK` tokens that do not occur in the training set. Such case may happen due to the small size of the datasets. The `UNK` tokens are explained more clearly in Section 3.3.2.

3.3.1 Preprocessing

Some preprocessing steps that we applied for all the above datasets were:

- Stripping all the grammatical function tags. So, nonterminals such as NP-SBJ, ADVP-TMP-CLR-TPC-1, and WHADVP=1 become NP, ADVP, and WHADVP respectively.
- Removing null elements such as *, *T*, and so forth from the parse tree. The complete specification of the null elements can be found in the Penn Treebank bracketing guidelines.

For Indonesian Treebank dataset, an additional step to remove extra parentheses around the terminal words was performed so the resulting parse trees looked like the English ones. As illustrations, after these preprocessing steps, a sample Penn Treebank parse tree

```
(S (NP-SBJ (PRP He)) (VP (MD ca) (RB n’t) (VP (-NONE- *?*))) (. .))
```

was transformed into

```
(S (NP (PRP He)) (VP (MD ca) (RB n’t)) (. .))
```

and the above Indonesian Treebank parse tree

```
(NP (NN (Kera))
    (SBAR (SC (untuk))
          (S (NP-SBJ (*))
             (VP (VB (amankan))
                 (NP (NN (pesta olahraga)))))))
```

was transformed into

```
(NP (NN Kera)
    (SBAR (SC untuk)
          (S (VP (VB amankan)
                 (NP (NN pesta_olahraga))))))
```

3.3.2 Unknown words

When training oracle files were generated, singleton words in the training set were converted into special `UNK` tokens for unknown words. For Penn Treebank datasets, we grouped the `UNK` tokens using the same grouping that Dyer et al. used in their implementation. These `UNK` tokens were grouped based on their characteristics, such as starting with a capital letter (`UNK-INITC`), lowercased (`UNK-LC`), lowercased and ending with *-ed* (`UNK-LC-ed`), numbers (`UNK-NUM`), and so forth. In total, there could be up to 50 of these tokens. For Indonesian Treebank datasets, however, we did not perform any grouping; we used only one such token (`UNK`).

3.3.3 Pretrained word embeddings

As explained in the previous section, our baselines include RNNGs equipped with pretrained word embeddings. We used pretrained embeddings trained on Wikipedia articles as described in (Bojanowski et al., 2016). The embeddings are available for both English and Indonesian and can be downloaded from their Github page[6]. These embeddings are 300-dimensional. The embeddings for English and Indonesian contain 2,519,370 and 300,686 pretrained word embeddings respectively.

3.4 Implementation

In this section, we elaborate our implementation details. This elaboration covers our RNNGs implementation which is based on the implementation by Dyer et al. (2016). We also cover how the discriminative and generative RNNGs are implemented differently in Dyer et al.'s, aside from the mathematical modeling differences discussed in Chapter 2. Next, we explain about the word clustering that is required by the class-factored softmax used in generative RNNGs. Lastly, we elaborate on how the class-factored softmax prevents the out-of-vocabulary problem from vanishing despite the incorporation of character embeddings.

3.4.1 Recurrent neural network grammars

Our RNNGs implementation extended the implementation by Dyer et al. (2016), which is available online[7]. In running our experiments, we used largely the same configuration as theirs, which is explained below. Note that we also explain some implementation details that are not explicitly mentioned in Dyer et al.'s paper but apparent from the implementation code.

[6] `https://github.com/facebookresearch/fastText/blob/master/pretrained-vectors.md`

[7] `https://github.com/clab/rnng`

3.4.1.1 Architecture

In Dyer et al.'s work, all the LSTMs, both in the discriminative and the generative variant of RNNGs, were fixed to have 2 layers. For the discriminative variant, the dimension of word embeddings, LSTMs inputs (except for the action history LSTM), and LSTMs hidden layers were set to 32, 128, and 128 respectively. As for the generative counterpart, they were all set to be 256-dimensional. In both variants, the action embeddings had 16 dimensions. We used this exact same architecture in our experiments.

3.4.1.2 Word embeddings

There are some differences between the discriminative and generative RNNGs in handling the word embeddings before feeding them as inputs to the LSTMs. For the discriminative RNNGs, a word embedding is first concatenated with its POS tag embedding and then passed through an affine transformation and activation function before it is fed as input to the LSTMs. This processing of POS tags follows the same approach described in (Dyer et al., 2015). However, for the generative RNNGs, POS tags are not used. Therefore, word embeddings are fed directly as inputs to the LSTMs without passing them through any affine transformation or activation function. The embeddings for POS tags are included as parameters, i.e. they are learned during training. The default size of these POS tag embeddings is 12, which was used by Dyer et al. We used the same size in our experiments.

3.4.1.3 Pretrained word embeddings

For the discriminative RNNGs, pretrained word embeddings are incorporated by concatenating them with the learned word embeddings, in addition to the POS tag embeddings as described previously. The concatenation result is then passed through an affine transformation and activation function as usual. For the generative RNNGs, however, Dyer et al. have not implemented pretrained word embeddings yet. Therefore, we wrote the implementation ourselves, based largely on how it is implemented for the discriminative counterpart. Essentially, whenever a word embedding is needed as input to the LSTMs, the learned word embedding and the pretrained word embedding are concatenated, and then passed through an affine transformation and activation function to get the final word embedding.

Recall that the dimension of word embeddings for the discriminative RNNGs is 32. This number is much smaller than the dimension of the pretrained embeddings which is 300. To account for this factor, we expanded the learned word embeddings for the discriminative RNNGs to be 300 as well when the pretrained word embeddings were employed.

3.4.1.4 Activation function

Although in their paper Dyer et al. describe that they used tanh activation function, in their code, all the activation functions are rectified linear units. A rectified linear unit (ReLU) activation function is defined as

$$\mathrm{relu}(x) = \max(0, x) \tag{3.4}$$

One well-known advantage of ReLUs as opposed to tanh functions is that ReLUs' gradients do not saturate for large values of x.

3.4.1.5 Training with `UNK` tokens

There is a difference on how `UNK` tokens are used in training between discriminative and generative RNNGs. Generative RNNGs use only the `UNK` tokens for its training, i.e. once a word is converted into an `UNK` token, they completely ignore what the actual word is during training since only the token is used. In contrast, discriminative RNNGs care what the actual word is because it has 50% probability of either using the actual word or its `UNK` token. This difference is not documented in their paper.

3.4.2 Word clustering

As explained in Section 2.3, generative RNNGs use class-factored softmax to model the word generation. This approach requires the words to be clustered so that the clusters can be used as the classes. Dyer et al. used Brown clustering algorithm (Brown et al., 1992) to obtain the clusters. We used the same clustering algorithm in our research. Since the clusters for PTB full dataset is provided in Dyer et al.'s implementation, we only need to run the algorithm on the other two datasets, PTB 1029 and ITB full, for each of the cross-validation fold. We used an implementation of Brown clustering that is available online[8].

3.4.3 Character embedding model

As explained previously in Section 2.4, we used addition model by Botha and Blunsom (2014) in our research. Our implementation is straightforward: whenever a word embedding is needed, the word is broken down to its characters and their embeddings are summed. The resulting vector is the embedding of the word. All the character embeddings have 32 dimensions.

Although we argued in Section 1.1 that using character embeddings alleviates the out-of-vocabulary problem, which means there is no need for `UNK` tokens anymore, we found that this is not the case because of the class-factored softmax in generative RNNGs. The softmax needs clusters of words to function, not characters. The clustering algorithm has to be trained on the `UNK`-ified words of the training set because if not, the class-factored softmax can encounter out-of-vocabulary words when the generative RNNGs are evaluated on the development or testing set. Also, since the generative RNNGs use the discriminative counterparts for inference, the discriminative variants also need to be trained on `UNK`-ified training set. Therefore, simply using character embeddings without modifying the class-factored softmax does not naturally solve the out-of-vocabulary problem.

To handle these `UNK` tokens when using character embeddings, our implementation separates the embeddings for the tokens. Essentially, when an embedding of a word is needed, the flow goes as follows:

1. If the word is not an `UNK` token, break down the word to its characters and compute the sum of their embeddings to get the word embedding.
2. Otherwise, the word is an `UNK` token. Obtain the embedding for the token from an embedding lookup table. No character-related operation is performed.

[8]Version 1.3: `https://github.com/percyliang/brown-cluster`

We chose to separate the two embeddings so that the character embeddings are actually estimated from the real words rather than artificial ones like the `UNK` tokens.

3.5 Experiments setup

This section describes the experiments setup we used in our research. Firstly, we discuss how training was done: initialization, optimizer, learning rate, and early stopping. Next, we describe the dropout regularization, how it is implemented for LSTMs, and more importantly how we chose our dropout rate. The section is then closed by our explanation on sampling for inference in generative RNNGs.

3.5.1 Training

Using Dyer et al.'s implementation, all RNNG parameters (including the biases) were initialized using *xavier initialization* (Glorot and Bengio, 2010). This initialization sets the parameter values to samples drawn from Uniform $\left(-\sqrt{\frac{6}{D}}, \sqrt{\frac{6}{D}}\right)$, where D is the sum of the parameter dimensions. Stochastic gradient descent was used as the optimizer with initial learning rate of 0.1. As the training progressed, the learning rate was decayed at each epoch according to the expression below

$$\eta_\tau = \frac{\eta_0}{1 + \tau r} \tag{3.5}$$

where η_τ is the learning rate at epoch τ, η_0 is the initial learning rate, and r is the decay rate. Similar to Dyer et al.'s experiments setup, we used $r = 0.05$ for the discriminative and $r = 0.08$ for the generative RNNGs. Unlike Dyer et al. though, we extended their implementation to use early stopping to train our parsers. Every 25 parameter updates, the trained parser was evaluated on the development set. The training ended if the parser's performance (F_1 score for the discriminative parser, likelihood for the generative parser) did not improve on the development set after 10 evaluations. We chose these values based on computational considerations. The training data was shuffled at the beginning of each epoch.

For our non-neural parsers, namely the Stanford factorized parser and Berkeley parser, we used their default settings when training. An exception is when we trained Stanford factorized parser on ITB full, we used left head finder instead of the default head finder optimized for Penn Treebank dataset. We did so because in Indonesian, the leftmost word is usually the head word (at least for noun phrases).

3.5.2 Dropout

Dyer et al. used dropout to regularize their LSTMs. In particular, they used RNN dropout regularization method proposed by Zaremba et al. (2015). This regularization applies dropout only on the outputs of the LSTM in each layer when they are fed as input to the next layer, including the inputs accepted by the LSTM itself. Dyer et al.'s implementation uses this dropout for every LSTM used by RNNGs: the stack LSTM, the buffer LSTM, the action history LSTM, and both forward and backward LSTMs of the syntactic composition function. We kept the same regularization procedure in our experiments.

Since we have different datasets (especially in terms of size) from Dyer et al., we had to tune our dropout rate. We performed grid search on multiples of 0.25, i.e. 0.0, 0.25, 0.5, and 0.75, and picked the dropout rate value which yielded the best performance on the development set. Note that we did not include dropout rate of 1.0 because this value results in all the vector values being set to zero. We tried these values rather than more fine-grained ones like multiples of 0.1 for computational reasons. Also, for PTB 1029 and ITB full datasets, we only tuned the dropout rate on the 0-th fold and applied the optimal value for all the folds rather than tuning each fold separately because of the same reason.

3.5.3 Sampling

Recall that Chapter 2 explains that sampling method is used for inference in the generative RNNGs. Dyer et al. introduced a flattening coefficient α when drawing samples from discriminative RNNGs. Concretely, each probability defined by the discriminative RNNGs was exponentiated by this coefficient α and then renormalized before samples were drawn. This flattening process acts similarly to a smoothing method because lower value of α results in a more uniform distribution and vice versa.

We also used this flattening coefficient in our experiments. We tuned this coefficient α on the development set by performing grid search on multiples of 0.25, i.e. 0.25, 0.5, 0.75, and 1.0. Note that we did not include 0.0 as this value results in a flat uniform distribution. Similar to dropout tuning, we chose these values rather than more fine-grained ones for computational reasons. We also tuned this coefficient α on only the 0-th fold of PTB 1029 and ITB full datasets and applied the optimal value for all folds. This procedure is the same as that of dropout rate tuning, and so is the reason. For every experiment, we drew 100 samples, the same value that Dyer et al. reported in their paper.

Chapter 4

Results and discussions

In this chapter, we report on the results of our experiments. This report includes the results from our baselines, both non-neural parsers and RNNGs. We first present the parsing F_1 scores on the test set for each parsers on every dataset. Next, we show a table of convergence speed of RNNGs in our experiments. Then, we show figures to illustrate how RNNGs work on small datasets. We then provide discussions on our presented results, including speculations on the interesting ones.

4.1 Results

Table 4.1 shows the parsing F_1 scores on the test set for all datasets. It shows the scores of our baseline parsers, namely the Stanford factorized parser (Klein and Manning, 2003), Berkeley parser (Petrov et al., 2006), original RNNGs by Dyer et al. (2016), and RNNGs equipped with pretrained word embeddings. It also shows the scores of our RNNGs with character embedding additional model by Botha and Blunsom (2014) employed. The original RNNGs' results were obtained by training the models using the same setup as Dyer et al., although the results are slightly lower than what they reported[1]. We trained the RNNGs with pretrained word embeddings with a slightly different early stopping configuration:

[1]Dyer et al. (2016) reported F_1 scores of 91.7 and 93.3 for the discriminative and generative model respectively. We obtained lower scores because we did not train the models long enough, due to the limited time we had.

Parser	PTB full	PTB 1029		ITB full	
		Cross-Validation	Fold 0	Cross-Validation	Fold 0
Stanford parser	81.31	64.60 (1.62)	65.25	61.41 (0.78)	61.51
Berkeley parser	90.02	71.05 (1.45)	72.50	65.73 (1.64)	68.32
Disc RNNG	*89.73	80.24 (1.53)	80.83	73.18 (0.52)	73.46
Gen RNNG	*92.30	83.00 (1.43)	83.79	**78.16** (0.45)	78.42
Disc RNNG +pretrained	90.00	n/a	80.92	n/a	74.91
Gen RNNG +pretrained	**92.71**	n/a	**83.98**	n/a	**78.63**
Disc RNNG +character	87.76	80.32 (1.91)	81.77	72.13 (1.28)	71.78
Gen RNNG +character	91.41	**83.36** (2.25)	83.93	76.40 (0.82)	76.71

Table 4.1: Test F_1 scores for all datasets. Stanford parser is by Klein and Manning (2003). Berkeley parser is by Petrov et al. (2006). Disc and gen refer to discriminative and generative variant of RNNG respectively. Suffix +pretrained and +character mean that pretrained word embeddings and character embedding addition model by Botha and Blunsom (2014) were employed respectively. Numbers inside parentheses are the standard deviations for the cross-validation results. n/a indicates that the experiment was too costly to run. Asterisk (*) means that the number is obtained by training plain RNNGs using the same setup as Dyer et al. (2016).

the model was evaluated on the development set every 100 parameter updates and training was stopped when the score did not improve after 5 evaluations. We used this configuration because of computational reasons.

From the table, we see that character embedding addition model does not seem to boost parsing performance for small datasets. However, looking closely, for most cases, the scores still overlap within one standard deviation. Therefore, they appear to have the same performance with plain RNNGs. Having said that, as the number of parameters in RNNGs with character embeddings is smaller, the training converged much faster. Table 4.2 shows the convergence speed for each RNNG model in our experiments.

From Table 4.2, we see that employing character embeddings seems to reduce training time for generative RNNGs. The reduction is especially profound for the large dataset, i.e. PTB full, where the plain generative RNNG needed 7,900 parameter updates to reach convergence while the one with character embedings

Parser	PTB full	PTB 1029		ITB full	
		Cross-Validation	Fold 0	Cross-Validation	Fold 0
Disc RNNG	3275	250	250	850	525
Gen RNNG	7900	100	100	175	160
Disc RNNG +pretrained	3600	n/a	800	n/a	325
Gen RNNG +pretrained	3900	n/a	100	n/a	155
Disc RNNG +character	1600	625	700	425	450
Gen RNNG +character	3200	75	75	175	175

Table 4.2: Convergence of training for different RNNG models. Convergence is defined as the number of parameter updates before the training is early-stopped. Disc and gen refer to the discriminative and generative variant respectively. Suffix +pretrained and +character mean that pretrained word embeddings and character embedding addition model by Botha and Blunsom (2014) were used respectively. The numbers shown for cross-validated datasets are the medians. n/a indicates that the experiment was too costly to run.

needed only 3,200. We see shorter training time for discriminative RNNGs as well, but interestingly on PTB 1029 dataset, character embeddings appear to impede convergence quite severely.

Our experiment results on the development set (available in Appendix A) show that plain RNNGs perform surprisingly well on small datasets. Upon observing this finding, we were interested in investigating how well RNNGs work if the training set size is reduced even more. Figure 4.1 show the test scores for the plain discriminative and generative RNNGs compared to the Stanford factorized parser and Berkeley parser when training set size was varied on the 0-th fold of PTB 1029 and ITB full dataset respectively. We experimented only with the 0-th fold to reduce the computational cost.

All our experiment results above were obtained using the configuration and experiment setups explained in the previous chapter, except for the RNNGs with pretrained word embeddings whose early stopping procedure was already mentioned above. The dropout rate and flattening coefficient α were tuned to the development set. This tuning was also performed for each level of training set size when we varied them. The tuned dropout rate and α values for all our experiments are available in Appendix B.

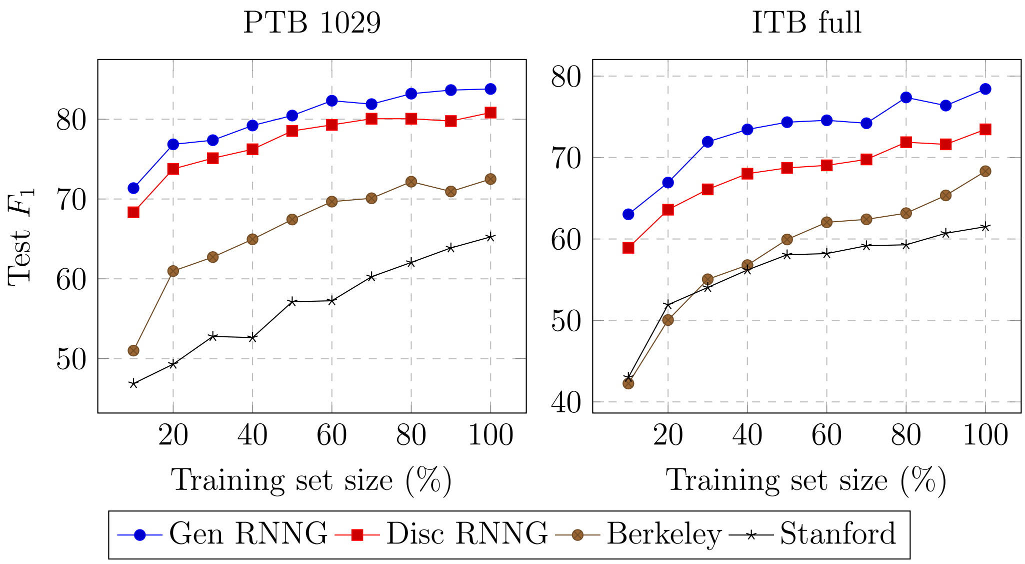


Figure 4.1: Test F_1 scores for the 0-th fold of small datasets with various training set sizes. Training set size of 10% corresponds to roughly 60 sentences. Disc and gen refer to the discriminative and generative variant of RNNGs respectively.

4.2 Discussions

Based on the results on Table 4.1, we observe that in general, RNNGs appear to perform better than non-neural parsers such as the Stanford factorized and Berkeley parser. One exception is the discriminative RNNG with character embedding addition model on PTB full dataset which has F_1 score of 87.76, lower than 90.02 of the Berkeley parser. Our proposed approach of employing character embedding model does not seem to help improve RNNGs' performance, even on the small datasets, despite the sparsity reduction it provides. Nevertheless, its results on the small datasets mostly lie within one standard deviation of the other RNNG baselines. Performance also drops on PTB full dataset, especially of the discriminative variant where the score drops roughly 2 points. We suspect that these performance drops are due to the addition model that ignores character ordering. Even in (Botha and Blunsom, 2014), they handled this issue by incorporating word embeddings in addition to morpheme embeddings. However, their solution will not result in sparsity reduction because the number of parameters will increase. Hence, we deliberately did not adopt their approach fully. Another interesting observation is that pretrained word embeddings improved performance on the large dataset (PTB full) but surprisingly did not do as much on the smaller ones (PTB 1029 and ITB full).

Although character embeddings do not appear to improve RNNGs' performance, they may help to reduce training time. Table 4.2 shows that RNNGs with character embeddings converge much faster than the ones without. The speedup is more profound on PTB full dataset, where both the discriminative and generative RNNGs are twice as fast to converge when character embeddings are employed. Even when compared to the RNNGs with pretrained word embeddings, character embeddings still converge faster. This trend can be observed on the small datasets as well, albeit not as profound. An exception to this trend is the discriminative RNNG on PTB 1029 which converged twice as slow compared to the plain counterpart. We are not sure as to why this phenomenon occurred.

As we explained before, seeing how plain RNNGs worked unexpectedly well on small datasets, we proceeded to see how they perform on various training set sizes. The results are shown in Figure 4.1. From the figure, we see that the best non-neural parser, namely the Berkeley parser, needed 50–60% training data to beat the discriminative RNNG trained on only 10% of the training set. For the generative RNNG, the disparity is even wider. Berkeley parser needed roughly 80% training data just to have equal performance as the generative RNNG with only 10% training data. This trend is observable on both PTB 1029 and ITB full. Therefore, it seems that RNNGs work really well and are robust on small data. This finding clashes with the preconception that neural networks need a lot of data to function well. We speculate that the shift-reduce parsing algorithm that RNNGs use contribute to their robust performance. At each step of the algorithm, RNNGs determine what action to take among only a handful number of possible actions. We believe this formulation ultimately results in a much smaller parameter space than estimating probabilities in PCFG parsers, which is what both Stanford factored and Berkeley parser essentially do.

Chapter 5

Conclusions and future work

In this chapter, we conclude our thesis by highlighting the contributions of this research, relating them with the motivations and objectives mentioned in Chapter 1. It is then closed by suggestions on some areas to which future research might be directed.

5.1 Conclusions

As stated in Section 1.1, there are several reasons that motivate this research. Firstly, most languages in the world, unlike English, are low-resource. There are not many, sometimes none at all, linguistic resources such as annotated treebank dataset that can be employed for natural language processing. We are interested to see if RNNGs work on such low-resource languages. We chose Indonesian as a sample of low-resource language. Secondly, character embeddings are found to be helpful for many language tasks, and at the same time mitigate the sparsity problem encountered when working on small data. The objectives of this research are also formulated around those motivations, namely the investigation of RNNGs in low-data scenario and the exploration of character embeddings for RNNGs. Based on the results and discussion in Chapter 4, we present the conclusions of this research.

Contrary to widespread presumption, RNNGs as neural network models, perform remarkably well on low-data scenario. Our experiments on small English and Indonesian datasets show that RNNGs can achieve decent performance when trained on even as low as 60 sentences. State-of-the-art non-neural parsers need 5–8 times more data to achieve similar level of performance. These findings suggest that RNNGs may be suitable to be used as constituency parsers for low-resource languages.

Using character embeddings does not seem to result in tremendous improvement on RNNGs' performance as initially hypothesized. This observation is true even for small datasets, where RNNGs supposedly benefit from the embeddings due to the sparsity reduction. Having said that, RNNGs with character embeddings converge faster, especially for large dataset which can be twice as fast as evidenced by our experiment results. They even converge faster than RNNGs with pretrained word embeddings. For some tasks, faster convergence might be more desirable, even if the performance drops slightly.

In summary, we have accomplished our research objectives. We investigated how RNNGs work on low-resource languages and found that they worked remarkably well, contrary to the presumption about neural network models on small data. We also explored the use of character embeddings, and discovered that they resulted in slight performance drop, but much faster convergence speed.

5.2 Future work

There are several aspects of this work that can still be refined for future research. Firstly, as noted in Chapter 3, we did not group the `UNK` tokens for Indonesian Treebank dataset. It is interesting if future work can do so and see how it impacts the results. Secondly, due to the limited time, we could not tune the learning rate that we used in stochastic gradient descent. Based on our experience, a good learning rate can improve performance drastically. We unintentionally re-trained one of our saved models once, and we obtained up to 6 points improvement in F_1 score on development set. This fact attests to the importance of learning rate tuning for future work. Thirdly, future research might be directed to modifying class-factored softmax used in generative RNNGs to be used jointly with character embeddings so that `UNK` tokens can truly be eliminated. This problem is what prevented the out-of-vocabulary problem from naturally vanishing as argued in Section 1.1. Fourthly, more complex character embedding models that are not order invariant might also be tried. Models based on convolutional networks (Kim et al., 2016) or bidirectional LSTMs (Ling et al., 2015) are both order invariant and more complex than addition model (Botha and Blunsom, 2014) that we used in this research.

Another area worth exploring for future work is to answer our speculation regarding the smaller parameter space of shift-reduce parsing in Section 4.2. It is interesting to train non-neural models, e.g. maximum entropy model, using the same shift-reduce parsing algorithm, and compare them to RNNGs. This comparison is fairer so the result can confirm if it is indeed the shift-reduce parsing algorithm that enables RNNGs to work well on small data or something inherent in RNNGs itself. Next, future work might also be directed to investigating why RNNGs with character embeddings converge slowly in small English dataset. Lastly, it is also appealing to see if RNNGs also work well with other low-resource languages.

Appendix A

Results on development set

Parser	PTB full	PTB 1029		ITB full	
		Cross-Validation	Fold 0	Cross-Validation	Fold 0
Stanford parser	80.33	64.50 (1.27)	62.25	61.48 (1.48)	61.66
Berkeley parser	89.11	70.83 (0.90)	71.54	65.68 (0.95)	65.15
Disc RNNG	88.74	80.89 (0.77)	80.98	74.05 (1.00)	73.69
Gen RNNG	91.29	83.47 (0.46)	83.26	78.09 (1.24)	78.32
Disc RNNG +pretrained	89.30	n/a	82.15	n/a	75.93
Gen RNNG +pretrained	92.15	n/a	83.68	n/a	79.38
Disc RNNG +character	86.84	81.17 (1.15)	81.27	73.54 (1.00)	73.57
Gen RNNG +character	90.39	82.98 (0.75)	82.65	76.81 (1.32)	77.21

Table A.1: Dev F_1 scores for all datasets. Stanford parser is by Klein and Manning (2003). Berkeley parser is by Petrov et al. (2006). Disc and gen refer to discriminative and generative variant of RNNG respectively. Suffix +pretrained and +character mean that pretrained word embeddings and character embedding addition model by Botha and Blunsom (2014) were employed respectively. Numbers inside parentheses are the standard deviations for the cross-validation results. n/a indicates that the experiment was too costly to run.

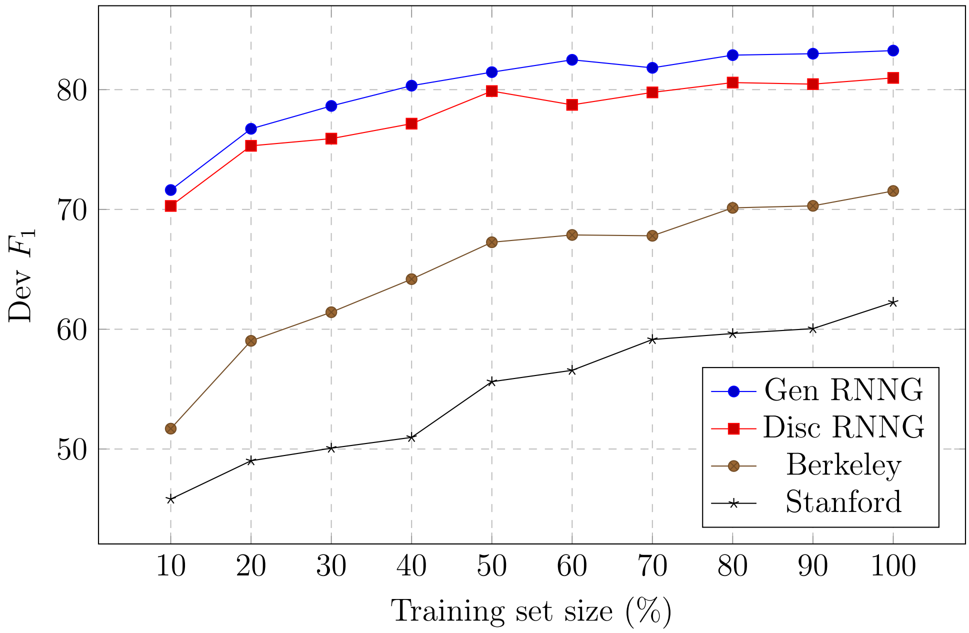


Figure A.1: Dev F_1 scores for the 0-th fold of PTB 1029 dataset with varying training set size. Training set size of 10% corresponds to roughly 60 sentences. Disc and gen refer to the discriminative and generative variant of RNNGs respectively.

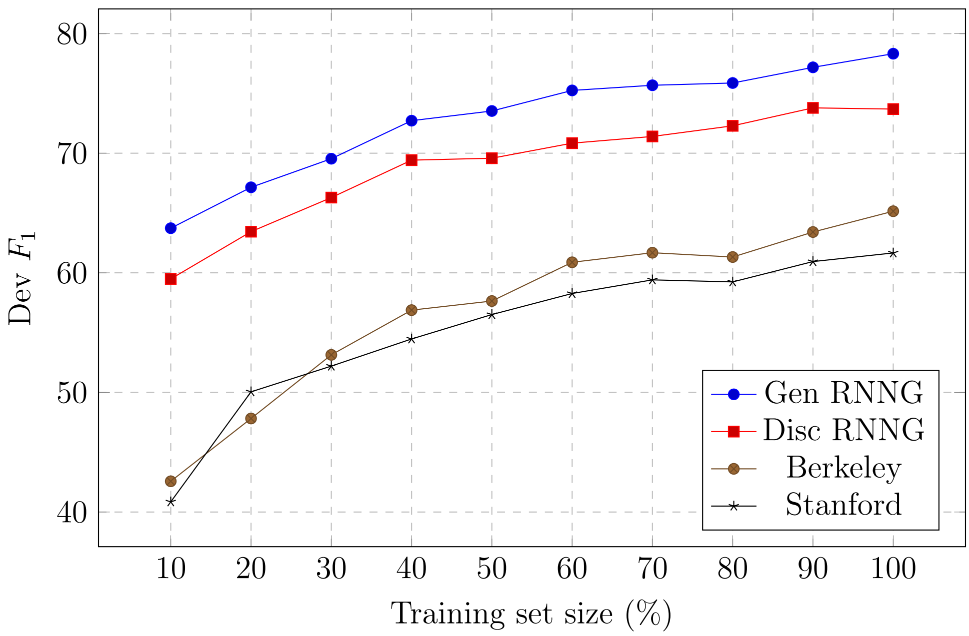


Figure A.2: Dev F_1 scores for the 0-th fold of ITB full dataset with varying training set size. Training set size of 10% corresponds to roughly 60 sentences. Disc and gen refer to the discriminative and generative variant of RNNGs respectively.

Appendix B

Tuning results

		PTB full	PTB 1029	ITB full
Disc RNNG	dropout rate	0.20	0.25	0.25
Gen RNNG	dropout rate	*0.30	0.25	0.50
	coefficient α	*0.80	1.00	0.50
Disc RNNG +pretrained	dropout rate	0.25	0.25	0.25
Gen RNNG +pretrained	dropout rate	0.25	0.50	0.50
	coefficient α	1.00	0.50	0.75
Disc RNNG +character	dropout rate	0.00	0.25	0.25
Gen RNNG +character	dropout rate	0.25	0.25	0.50
	coefficient α	1.00	0.75	0.50

Table B.1: Optimal dropout rate and flattening coefficient α obtained by tuning on the development set for each dataset. For PTB 1029 and ITB full datasets, the tuning was done only on the 0-th fold's development set. Disc and gen refer to discriminative and generative variant of RNNG respectively. Suffix +pretrained and +character mean that pretrained word embeddings and character embedding addition model by Botha and Blunsom (2014) was employed respectively. Asterisk (*) indicates that the number is obtained from (Dyer et al., 2016)

Size (%)	Disc RNNG	Gen RNNG	
	dropout rate	dropout rate	coefficient α
10	0.25	0.50	1.00
20	0.25	0.50	1.00
30	0.50	0.50	1.00
40	0.25	0.50	0.50
50	0.25	0.50	0.75
60	0.25	0.50	0.50
70	0.25	0.75	0.75
80	0.25	0.50	0.75
90	0.50	0.50	1.00

Table B.2: Optimal dropout rate and flattening coefficient α obtained by tuning on the development set of PTB 1029 dataset for each level of training set size. Disc and gen refer to the discriminative and generative variant of RNNG respectively.

Size (%)	Disc RNNG	Gen RNNG	
	dropout rate	dropout rate	coefficient α
10	0.25	0.50	0.50
20	0.25	0.50	1.00
30	0.25	0.50	0.75
40	0.50	0.50	1.00
50	0.50	0.75	1.00
60	0.25	0.75	1.00
70	0.25	0.50	1.00
80	0.25	0.50	0.50
90	0.25	0.50	0.75

Table B.3: Optimal dropout rate and flattening coefficient α obtained by tuning on the development set of ITB full dataset for each level of training set size. Disc and gen refer to the discriminative and generative variant of RNNG respectively.

Bibliography

Ballesteros, M., Dyer, C., and Smith, N. A. (2015). Improved transition-based parsing by modeling characters instead of words with LSTMs. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 349–359, Lisbon, Portugal. Association for Computational Linguistics.

Baltescu, P. and Blunsom, P. (2015). Pragmatic neural language modelling in machine translation. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 820–829, Denver, Colorado. Association for Computational Linguistics.

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166.

Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. (2016). Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*.

Botha, J. A. and Blunsom, P. (2014). Compositional morphology for word representations and language modelling. In *Proceedings of the 31st International Conference on Machine Learning*, volume 32, pages 1899–1907, Beijing, China. PMLR.

Brown, P. F., Desouza, P. V., Mercer, R. L., Pietra, V. J. D., and Lai, J. C. (1992). Class-based n-gram models of natural language. *Computational linguistics*, 18(4):467–479.

Cooijmans, T., Ballas, N., Laurent, C., Gülçehre, Ç., and Courville, A. (2016). Recurrent batch normalization. *arXiv preprint arXiv:1603.09025*.

dos Santos, C. N. and Zadrozny, B. (2014). Learning character-level representations for part-of-speech tagging. In *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 1818–1826, Beijing, China. PMLR.

Dyer, C., Ballesteros, M., Ling, W., Matthews, A., and Smith, N. A. (2015). Transition-based dependency parsing with stack long short-term memory. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 334–343, Beijing, China. Association for Computational Linguistics.

Dyer, C., Kuncoro, A., Ballesteros, M., and Smith, N. A. (2016). Recurrent neural network grammars. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 199–209, San Diego, California. Association for Computational Linguistics.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14(2):179–211.

Gildea, D. and Jurafsky, D. (2002). Automatic labeling of semantic roles. *Computational Linguistics*, 28(3):245–288.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Teh, Y. W. and Titterington, M., editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy. PMLR.

Goodman, J. (2001). Classes for fast maximum entropy training. In *Proceedings of the 2001 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pages 561–564.

Hochreiter, S. (1991). *Untersuchungen Zu Dynamischen Neuronalen Netzen.* Diploma thesis, TU Munich.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Jurafsky, D. and Martin, J. H. (2009). *Speech and Language Processing.* Prentice Hall, 2nd edition.

Kim, Y., Jernite, Y., Sontag, D., and Rush, A. M. (2016). Character-aware neural language models. In *Proceedings of the 2016 Conference on Artificial Intelligence (AAAI)*.

Klavans, J., Liberman, M., Marcus, M., Roukos, S., Santorini, B., Strzalkowski, T., Abney, S., Flickenger, S., Gdaniec, C., Grishman, C., Harrison, P., Hindle, D., Ingria, R., and Jelinek, F. (1991). Procedure for quantitatively comparing the syntactic coverage of English grammars. In Black, E., editor, *Proceedings of the Workshop on Speech and Natural Language*, HLT '91, pages 306–311, Pacific Grove, California. Association for Computational Linguistics.

Klein, D. and Manning, C. D. (2003). Fast exact inference with a factored model for natural language parsing. In Becker, S., Thrun, S., and Obermayer, K., editors, *Advances in Neural Information Processing Systems 15*, pages 3–10. MIT Press.

Kuncoro, A., Ballesteros, M., Kong, L., Dyer, C., Neubig, G., and Smith, N. A. (2017). What do recurrent neural network grammars learn about syntax? In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, pages 1249–1258, Valencia, Spain. Association for Computational Linguistics.

Lee, J., Cho, K., and Hofmann, T. (2016). Fully character-level neural machine translation without explicit segmentation. *arXiv preprint arXiv:1610.03017*.

Ling, W., Dyer, C., Black, A. W., Trancoso, I., Fernandez, R., Amir, S., Marujo, L., and Luis, T. (2015). Finding function in form: Compositional character models for open vocabulary word representation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1520–1530, Lisbon, Portugal. Association for Computational Linguistics.

Petrov, S., Barrett, L., Thibaux, R., and Klein, D. (2006). Learning accurate, compact, and interpretable tree annotation. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th Annual Meeting of the Association for Computational Linguistics*, ACL-44, pages 433–440, Sydney, Australia. Association for Computational Linguistics.

Punyakanok, V., Roth, D., and Yih, W.-t. (2008). The importance of syntactic parsing and inference in semantic role labeling. *Computational Linguistics*, 34(2):257–287.

Vylomova, E., Cohn, T., He, X., and Haffari, G. (2016). Word representation models for morphologically rich languages in neural machine translation. *arXiv preprint arXiv:1606.04217*.

Zaremba, W., Sutskever, I., and Vinyals, O. (2015). Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*.