University of Potsdam

Faculty of Human Sciences

Bachelor Thesis

in Computational Linguistics

Submitted in Fulfillment of the Requirements for the Degree of Bachelor of Science

Topic: Investigating the ability of RNN architectures to learn context-free grammars by example of Dyck(2)

Author: Fynn Dobler <fynndobler@gmail.com>
Matr.-Nr. 775710

1. Supervisor: Dr. Thomas Hanneforth
2. Supervisor: Dr. Uladzimir Sidarenka

Summary

The capability of three major recurrent neural network (RNN) architectures - SRNN, LSTM and GRU - to learn the underlying structure of the Dyck(2) grammar has been investigated. To assess the influence of such factors as model complexity and training corpus composition, each architecture was instantiated in 27 models, with a model containing n hidden units ($n = \{2^1, 2^2, \ldots, 2^9\}$) each being trained on a character level on one of three corpora. In total, 81 models were trained and tested. The corpora consisted of a Baseline corpus to compare to, as well as a corpus containing an increased frequency for long-range dependencies (High LRD) and one with a decreased frequency thereof (Low LRD). The models were evaluated on two classification experiments, designed to assess handling of long-range dependencies and increasing nesting levels. The results were reported in terms of accuracy, precision, recall and F1 score, as well as a closer look at the distribution of false positives. The findings show the models as likely to model Dyck(2) if they were trained on the Low LRD corpus, with LSTMs and GRUs achieving promising results.

Abstract

Formal grammars, specifically context-free grammars (CFGs), are powerful tools with which to model natural languages. In this thesis, the capability of several recurrent neural networks (RNNs) to learn CFGs by proxy of Dyck(2) was investigated. The impact of training corpus composition was assessed by training models on three different corpora of varying complexity. To assess whether Dyck(2) was learned, the networks classified words into belonging or not belonging to Dyck(2) in two experiments desgined to test their ability to generalize to both extreme long-range dependencies (LRDs) and unseen nesting depths (NDs). Only few RNNs achieved above-chance accuracy. For the ones that did, low training data complexity facilitated generalization, while high complexity showed an inhibitive effect.

Zusammenfassung

Die Fähigkeit von drei bekannten RNN Architekturen - SRNN, LSTM und GRU - die unterliegende Struktur der Dyck(2)-Grammatik zu lernen wurde untersucht. Die Faktoren von Modellkomplexität und Korpuskomposition wurden berücksichtigt, indem zu jeder Architektur je 27 Modelle trainiert worden sind - je ein Modell mit n versteckten Einheiten ($n = \{2^1, 2^2, \ldots, 2^9\}$) wurde auf einem von drei Korpora trainiert. Insgesamt wurden 81 Modelle trainiert und getestet. Die drei Korpora waren ein Baseline Korpus, zu dem die anderen verglichen wurden. Zusätzlich wurde ein Korpus kreiert, in dem sich Abhängigkeiten zwischen einzelnen Buchstaben häufig lange erstrecken (High LRD), sowie ein Korpus wo diese Abhängigkeiten häufig kurz sind (Low LRD). Die Modelle wurden auf zwei verschiedenen Klassifikations-Experimenten evaluiert, welche darauf ausgelegt waren, das Modellverhalten bei langen Abhängigkeiten und tiefer Rekursion zu untersuchen. Die Ergebnisse wurden in Form von Genauigkeit, Precision, Recall und F1 Score angegeben. Zusätzlich wurde die Verteilung von falsch als korrekt klassifizierten Wörtern betrachtet. Die Komplexität des Trainingkorpus' hat sich als großer Einfluss auf die Lernfähigkeit von Modellen erwiesen: Modelle, die auf dem Low LRD Korpus trainiert worden sind, waren erfolgreicher als die Baseline Modelle, während High LRD Modelle weniger erfolgreich waren.

Contents

List of Figures

List of Tables

Index of Listings

1 Introduction

In the 2010s, seemingly every major technology company developed its own "personal assistant" system, a program that allows the end-user to interact with the company's services more intuitively by interpreting spoken natural language commands. Apple's Siri, Amazon's Alexa and Google's succinctly named Assistant have been irrevocably ingrained in day-to-day life. While the ethical and data security concerns raised by this development are still a point of contention, it is clear that Natural Language Processing (NLP) applications have boomed from a niche field to a rapidly growing multi-million dollar industry[1]. Despite state-of-the-art performance on NLP tasks such as machine translation, text classification, sentiment analysis and speech recognition having made leaps and bounds in the past decade, these systems are still far from acquiring a perfect understanding of natural language. Recently, many new Recurrent Neural Network (RNN) model ideas have been experimented with, like the Clockwork RNN (Koutník et al. (2014)) or the Recurrent Unit with a Stack-like State (RUSS) (Bernardy (2018)), often designed to excel at a specific task. To showcase the new model's superiority, its performance on a task is usually compared to that of a more well-known architecture, such as the Simple RNN (SRNN), the Long Short Term Memory (LSTM) or the Gated Recurrent Unit (GRU).

What is missing from the current state of literature is, however, a robust comparison of these three architectures on a task that adequately showcases their respective ability to perform well on natural language data. I seek to fill that gap with my work by trying to answer the following questions:

1. Can an SRNN, LSTM or GRU architecture learn the Dyck language with two pairs of brackets (D_2)?

2. If they cannot, what poses the highest difficulty in doing so?

3. What influence, if any, does corpus construction have on model performance, specifically generalizability?

The following work consists of five main parts, each of which will be briefly summarized hereunder:

In Chapter 2, I introduce the core concepts relevant for this thesis: formal languages, the complexity of Natural Language, Dyck languages, three neural network architectures and an overview of related works to contextualize my work within the current state of research. I describe model design and training, corpus construction and the two experiments I conduct in this work in Chapter 3. I report the results for each of the experiments in Chapter 4 and discuss them in detail in Chapter 5. Finally,

[1] https://www.tractica.com/newsroom/press-releases/natural-language-processing-market-to-reach-22-3-billion-by-2025/

I seek to answer the research questions posed above with my experimental results in Chapter 6 and suggest further avenues of research on this topic.

2 Theoretical Background

2.1 Formal Languages and Formal Grammars

A formal language $L(G)$ is defined as a subset of all words Σ^* over an alphabet Σ, where all words need to comply with the formal grammar G. As per Jurafsky and Martin (2009), the definition of a formal grammar is $G = \{N, \Sigma, R, S\}$, where N is a set of non-terminal symbols, Σ is a set of terminal symbols (alphabet), R is a set of rules of the form $\alpha \rightarrow \beta$ (where α and β are strings of symbols from $(\Sigma \cup N)^*$) and S is a designated start symbol. Following the two definitions, $L(G)$ consists of all strings w that can be derived from the start symbol S in a finite number of steps, formally $\{w \in \Sigma^* | S \overset{*}{\underset{\mathrm{G}}{\Rightarrow}} w\}$. As such, a word $w \in \Sigma^*$ that cannot be derived from S in a finite number of steps is not part of $L(G)$.

Formal grammars differ in terms of complexity and can be described in a hierarchical manner. Grammars of higher complexity have a greater generative power than grammars of lower complexity. The most commonly used hierarchy of grammars is the Chomsky hierarchy (Chomsky (1959)). In this hierarchy, formal grammars are classified into four types, sorted from most powerful to least powerful: Turing equivalent (Type 0), Context Sensitive (Type 1), Context Free (Type 2) and Regular (Type 3). The difference in generative power and complexity stems from increasing restrictions imposed on the rules of the grammar - a Type 3 grammar is more restrictive than a Type 0 grammar. As such, every grammar of a higher type is a subset of the previous type of grammar. A visual representation of this property can be found in Figure 1.

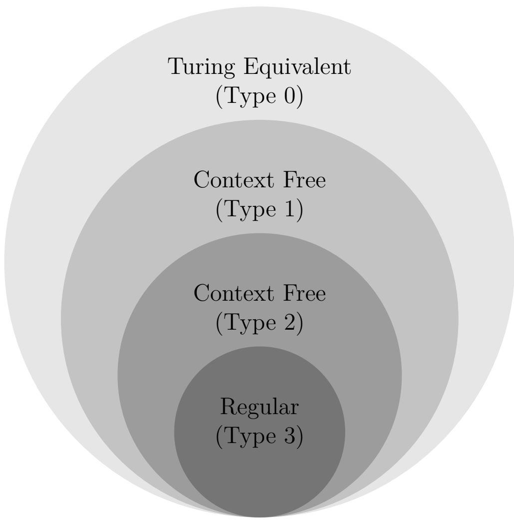

Figure 1: A visual representation of the Chomsky Hierarchy.

The four types of formal grammars can be defined by the form their rules can take. An overview over these rules as per Jurafsky and Martin (2009) can be found in Table 1, where A is a single non-terminal, α, β, γ are strings of terminal and non-terminal symbols, and x is a string of terminal symbols. α, β and γ may be empty unless specifically disallowed. The table is supplemented with a column describing the corresponding automaton capable of accepting or recognizing the grammar.

2.2 Formal Grammars and Natural Language

The correspondence of formal grammars to automata (i.e. Kleene's Theorem for regular languages and finite automata) and Computational Complexity Theory lends itself to consider natural languages under the same lense. While formal grammars constitute powerful tools with which phenomena in natural language can be modelled, assessing the precise complexity of Natural Language is the subject of ongoing investigation (Fitch et al. (2012), Petersson and Hagoort (2012), Newmeyer and Preston (2014)). Arguments answering that question usually seek to establish lower bounds: If there is a phenomenon in a natural language that cannot be described with a given type of grammar, natural language must be - however slightly - more complex than that type allows. Such arguments increase in credibility the more frequently they can be replicated for phenomena in multiple languages. The arguments establishing natural languages as supra-context-free (i.e. more complex than CFGs) as well as contrary evidence from empiric research shall be presented here.

2.2.1 Natural Language as supra-regular

English, as well as several other languages (Hagège (1976)) allow for center embedding, the embedding of a phrase into another phrase of the same type.

(1) The man eats.

(2) The man the boss fired eats.

(3) The man the boss the investor distrusted fired eats.

Type	*Name*	*Rule Skeleton*	*Automaton*
0	Turing Equivalent	$\alpha \to \beta$, s.t. $\alpha \neq \epsilon$	Turing Machine (recognized)
1	Context Sensitive	$\alpha A \beta \to \alpha \gamma \beta$, s.t. $\gamma \neq \epsilon$	Linear Bound Automata (accepted)
2	Context Free	$A \to \gamma$	Push Down Automata (accepted)
3	Regular	$A \to xB$ or $A \to x$	Finite-State Automata (accepted)

Table 1: Overview of formal grammar properties according to Jurafsky and Martin (2009), augmented with corresponding automata.

(4) The man the boss the investor the police investigated distrusted fired eats.

Let the set E contain all grammatical sentences of English, and let the noun phrases and transitive verbs constitute following sets:

$$A = \{\text{the boss, the investors, the police}, \dots\}$$
$$B = \{\text{fired, distrusted, investigated}, \dots\}$$

Then the following two sets can be defined.

$$E' = \{\text{the man } a^n b^n \text{ eats} \mid n \geq 0\}$$
$$R = \{\text{the man } a^* b^* \text{ eats}\}$$

a^n and b^n are finite sequences of size n of elements of sets A and B, respectively. E' describes a subset of E, namely $E \cap R$. Since regular languages are closed under intersection and E' is not regular, E is not regular.[2]

While this proof is correct under the framework of Formal Language Theory, the validity of claiming that it shows natural language to be supra-regular is debatable. Research in psycholinguistics shows that native speakers have faced severe problems processing center embeddings of depth two or higher, yielding long processing times, an incomplete understanding of the presented sentence or leading the participants to judge the sentence as ungrammatical (Hamilton and Deese (1971), Frank et al. (2016)). Furthermore, the corpus-driven analysis by Karlsson (2007) suggests an upper limit of center embedding depth three in the seven investigated languages.

2.2.2 Natural Language as supra-context-free

Similarly to the proof given in Section 2.2.1, an argument characterizing natural language as supra-context-free can be brought forth. It is based on embedded infinitival verb phrases found in Swiss German (Shieber (1987)).

(5) Jan säit das mer em Hans es huus haend wele hälfe
Jan said that we the Hans-DAT the house-ACC have wanted help
aastriiche.
paint
'Jan said that we have wanted to help Hans paint the house.'

(6) Jan säit das mer d'chind em Hans es huus haend
Jan said that we the children-ACC the Hans-DAT the house-ACC
wele laa hälfe aastriiche.
have wanted let help paint
'Jan said that we have wanted to let the children help Hans paint the house.'

[2]The proofs for regular languages being closed under intersection and E' not being regular can be found in Hopcroft et al. (2006) and Sipser (2013).

Four finite sets can be constructed from these examples: accusative noun phrases ($A = \{$d'chind, $\dots\}$), dative noun phrases ($B = \{$em Hans, $\dots\}$), verbs taking accusative objects ($C = \{$laa, $\dots\}$) and verbs taking dative objects ($D = \{$hälfe, $\dots\}$). Let the set S then be the set of all grammatical sentences of Swiss German. Again, the two following sets can be defined:

$$S' = \{\text{Jan säit das mer } a^n b^m \text{ es huus haend wele } c^n d^m \text{ aastriiche} \mid n, m \geq 0\}$$
$$R = \{\text{Jan säit das mer } a^* b^* \text{ es huus haend wele } c^* d^* \text{ aastriiche}\}$$

S' is not context-free and results from $S \cap R$. Since context-free sets are closed under intersection with regular sets, G cannot be context-free.[3]

Curiously enough, empirical research into the matter of processing similar cross-serial dependencies in Dutch suggests them to be generally easier to process than nested dependencies (i.e. the ones used to prove natural language to be supra-regular) (Bach et al. (1986)).

2.3 Dyck Languages

Whether natural language is regular, context-free, supra-regular or supra-context-free is a distinction of only tangential relevance for this work. The first two cases are fully covered by CFGs, while the other two leave room for some natural language productions outside of the scope of CFGs. The characteristics of supra-context-free examples in natural language show a *weak* non-context-freeness, making CFGs sufficient for covering the vast majority of natural language productions. With this assumption, an appropriate CFG for a model to learn must be found. The most important property of this grammar is that model performance on its language must allow for strong conclusions about the learnability of any other CFG. In doing so, one can make reasoned assumptions about potential model performance on natural language data.

One such grammar is the Dyck Grammar, which can produce an array of Dyck Languages. Let $D_n = \{N, \Sigma, R, S\}$ with

$$
\begin{aligned}
N &= \{S\} \\
\Sigma &= \{\epsilon, O_1, O_2, \dots, O_n, C_1, C_2, \dots, C_n\} \\
R &= \{ \\
&\qquad S \rightarrow \epsilon \\
&\qquad S \rightarrow SS \\
&\qquad S \rightarrow O_n\, S\, C_n\, \},
\end{aligned}
$$

[3]The respective proofs for S' not being context-free and context-free sets being closed under intersection with regular sets can be found in Hopcroft et al. (2006) and Sipser (2013).

where O_n represents an opening parenthesis, C_n represents a closing parenthesis and n denotes the number of distinct pairs of parentheses. D_1, then, denotes the Dyck Language with $\Sigma = \{\epsilon, (,)\}$, D_2 the Dyck Language with $\Sigma = \{\epsilon, (, [,],)\}$, et cetera.

Within the family of Dyck Languages, D_2 is of particular interest. According to the Chomsky-Schützenberger Representation Theorem (Chomsky and Schützenberger, 1963), for every context-free language L there exists a positive integer n, a regular language R, and a homomorphism h so that $L = h(D_n \cap R)$. Following the proof in Autebert et al. (1997), a homomorphism g_n can be constructed so that $D_n = g_n^{-1}(D_2)$. It follows that every context-free language can be represented as $L = h(g_n^{-1}(D_2) \cap R)$. As such, every CFL could be represented via homomorphisms on D_2 and intersections with a regular language. Assuming natural languages to be context-free and bearing in mind that using a formal language is a choice of abstraction which allows for precise control over corpus composition, this makes D_2 the language of choice when comparing neural network performance.

2.4 Neural Network Architectures

2.4.1 Simple RNN

Recurrent Neural Networks (RNNs) (Elman (1990)) are a neural network architecture particularly suited to processing sequential information by design: the RNN's output at a time step t is fed back as its input at the following time step $t + 1$. Not only does this enable RNNs to process sequences of arbitrary length, it also makes every output dependent on the previous computation as well as the current input. This property equips RNNs with a "memory" for previous inputs, allowing them to capture context dependencies a context-agnostic model cannot adequately learn.

Within the frame of this work, the specific case of the Simple RNN (SRNN) is considered. It is a three layer networks, consisting of an input layer, a hidden layer and an output layer. The hidden state h_t at time step t given the input vector x_t and the output vector y_t are calculated as per the following equations:

$$h_t = f(\boldsymbol{W_{xh}} x_t + \boldsymbol{W_{hh}} h_{t-1}) \tag{1}$$

$$y_t = \boldsymbol{W_{hy}} h_t \tag{2}$$

The function f constitutes a non-linear transformation, like tanh or ReLU. $\boldsymbol{W_{xh}}$, $\boldsymbol{W_{hh}}$, $\boldsymbol{W_{hy}}$ are matrices of the weights connecting the input layer to the hidden layer, the hidden layer to itself and the hidden layer to the output layer, respectively.

When training RNNs, it is beneficial to think of the network as unfolding into an architecture with one layer per time step. A visualisation is provided in Figure 2. These conceptual layers share their parameters - if any weight changes at time step t, the weight also changes at $t+1, t+2, \ldots, t+n$. Isolated changes are not possible. A popular training algorithm for RNNs is Backpropagation Through Time (BPTT) (Williams and Zipser (1995)), a gradient based algorithm designed for recurrent rather than feedforward networks. However, as Bengio et al. (1994) and Hochreiter (1998) show, RNNs suffer from a fundamental flaw: the aptly named vanishing gradient problem, in which the training gradient diminishes to zero throughout the layers.

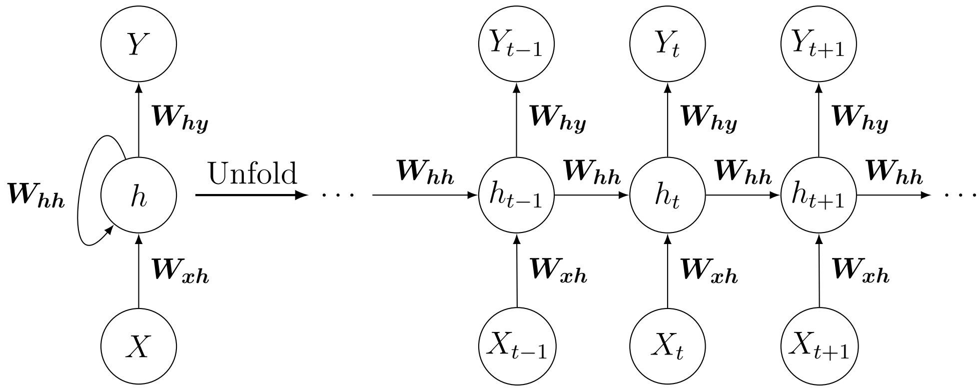

Figure 2: An RNN, unfolded through time.

2.4.2 LSTM

Long-Short Term Memory networks (LSTM) were designed by Hochreiter and Schmidhuber (1997) as an RNN architecture which preserves the RNN capabilities of processing sequential data of arbitrary length and capturing context dependencies, while circumventing the vanishing gradient problem.

LSTMs are based on self-connected linear units which are regulated by three gates consisting of a sigmoid layer σ each: input (in), output (out) and forget (forget). At every time step, the concatenated vector of the previous hidden state h_{t-1} and the current input x_t are received by all three gates. The sigmoid layer transforms every value in the concatenated vector to a value in range $[0, \ldots, 1]$ - a 0 translates to forgetting the information, while a 1 passes it through completely. Thus, the output of the gates determines what information is let through the input gate, passed through the output gate or forgotten by the self-connected linear unit.

$$\begin{aligned}
\mathrm{in}_t &= \sigma_{\mathrm{in}}(\boldsymbol{W}_{\mathrm{in}} \cdot [h_{t-1}, x_t] + b_{\mathrm{in}}) \\
\mathrm{out}_t &= \sigma_{\mathrm{out}}(\boldsymbol{W}_{\mathrm{out}} \cdot [h_{t-1}, x_t] + b_{\mathrm{out}}) \\
\mathrm{forget}_t &= \sigma_{\mathrm{forget}}(\boldsymbol{W}_{\mathrm{forget}} \cdot [h_{t-1}, x_t] + b_{\mathrm{forget}})
\end{aligned}$$

Finally, the cell state C_{t-1} is updated to C_t and h_t is set.

$$\begin{aligned}
C_t &= \mathrm{forget}_t \odot C_{t-1} + \mathrm{in}_t \odot \tanh(\boldsymbol{W_C} \cdot [h_{t-1}, x_t] + b_{\mathrm{C}}) \\
h_t &= \mathrm{out}_t \odot \tanh(C_t)
\end{aligned}$$

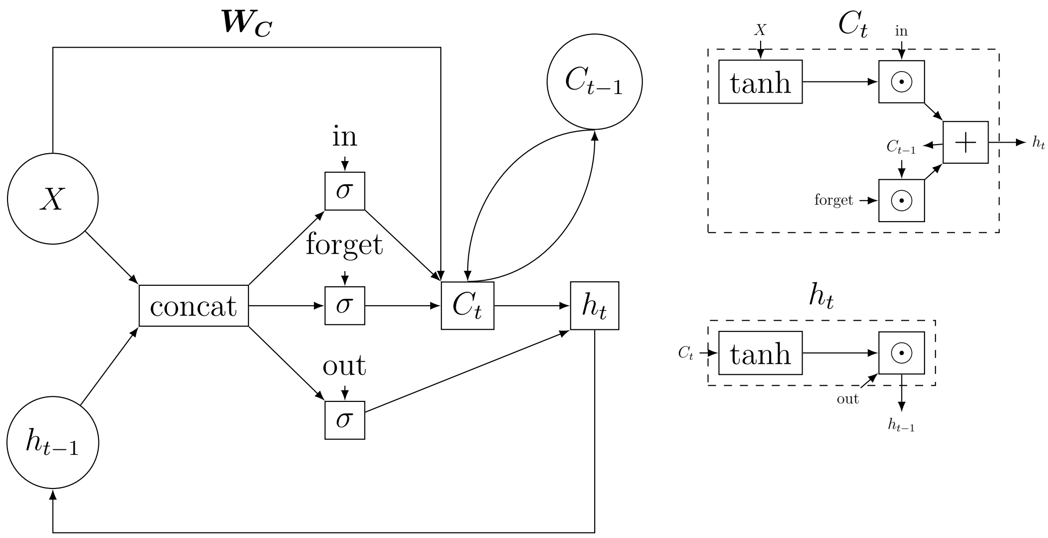

Figure 3: An LSTM memory cell.

2.4.3 GRU

A less complex alternative to LSTMs, the Gated Recurrent Unit (GRU) was developed by Cho et al. (2014). The information flow within the GRU is handled by just two gates: reset (r) and update (z). The update gate determines how much information from previous time steps is passed along for further time steps, while the reset gate enables the model to drop irrelevant information and only consider the current input rather than the previous hidden state, as described in the equations below, where j is the j-th hidden unit, σ is the squashing sigmoid function, $\boldsymbol{W}$ and $\boldsymbol{U}$ are learned gate-dependent weight matrices and ϕ is a non-linear function.

$$
\begin{aligned}
r_j &= \sigma\Big([\boldsymbol{W_r}x]_j + [\boldsymbol{U_r}h_{t-1}]_j\Big) \\
z_j &= \sigma\Big([\boldsymbol{W_z}x]_j + [\boldsymbol{U_z}h_{t-1}]_j\Big) \\
h_j^t &= z_j h_j^{t-1} + (1 - z_j)\tilde{h}_j^t \\
\tilde{h}_j^t &= \phi\Big([\boldsymbol{W}x]_j + [\boldsymbol{U}(r \odot h_{t-1})]_j\Big)
\end{aligned}
$$

2.5 Related Works

Currently, NLP models are trained and tested on vast datasets, such as the CoNLL Shared Tasks. By evaluating a variety of different models and approaches on the same data, it is possible to easily assess which one poses the current state-of-the-art for any given NLP task.

The earliest research on neural network architectures was done before such datasets were widely available and easy to process (Cleeremans et al. (1989), Elman (1990),

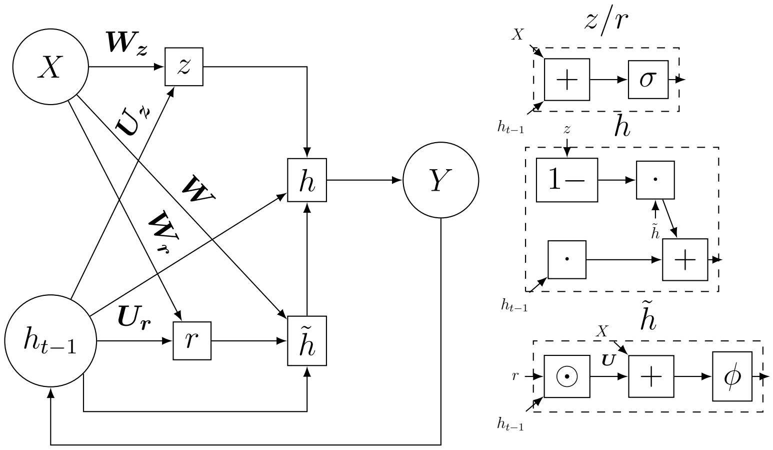

Figure 4: Illustration of a GRU.

Paper	D_n	*Grammar Probability*	*Training Corpus Size*
Deleu and Dureau (2016)	1	equal	unclear
Bernardy (2018)	1, 5	equal	$102,400$
Li et al. (2018)	1	modified	200 - $20,000$
Skachkova et al. (2018)	1-5	modified	$131,072$
Sennhauser and Berwick (2018)	2	modified	$1,000,000$
Suzgun et al. (2019)	1-2	modified	$10,000$
Yu et al. (2019)	2	modified	$1,000,000$

Table 2: Overview of corpus sizes in current works.

Zeng et al. (1994), Hochreiter and Schmidhuber (1997), Rodriguez and Wiles (1998), Gers and Schmidhuber (2001)). During that time, novel architectures and algorithms were mostly scored on formal language datasets, with the test set containing longer words than the training set to assess learning success. However, evaluating on formal languages comes with its own advantages and challenges.

Primarily, it is undeniably cheaper than scoring on a natural language dataset. By deriving words from the grammar, datasets of arbitrary length with arbitrary properties can be generated. However, performance on a formal grammar dataset should always be understood as a simplified benchmark. As mentioned in Sections 2.2.1 and 2.2.2, the formal complexity of natural language is debatable, limiting the significance of formal benchmark performance for NLP tasks. Nevertheless, formal language datasets are still used to evaluate the performance of novel architectures to this day, as done by Joulin and Mikolov (2015), Bernardy (2018), Deleu and Dureau (2016), Li et al. (2018) and Yu et al. (2019).

In addition to the exploration of new architectures, formal languages are also still used to investigate particular behaviours of well-established architectures, such as LSTMs (Sennhauser and Berwick (2018)), or to compare several established models on a specific set of tasks (Skachkova et al. (2018), Suzgun et al. (2019)).

Paper	*Accuracy*	*Perplexity*	*Cell State*	*AUC*	*Error Rate*
Deleu and Dureau (2016)	No	No	No	Yes	No
Bernardy (2018)	Yes	No	No	No	No
Skachkova et al. (2018)	Yes	Yes	No	No	No
Sennhauser and Berwick (2018)	No	No	Yes	No	Yes
Suzgun et al. (2019)	Yes	No	Yes	No	No
Yu et al. (2019)	No	Yes	No	No	Yes

Table 3: Overview of reported values for performance. Cell State Analysis does not refer to a unified method, it merely means the paper investigates cell states at all. AUC refers to the area under the curve for an increasing length of Dyck words the model was able to generalize.

Paper	*Architectures*
Deleu and Dureau (2016)	Neural Turing Machine, LSTM
Bernardy (2018)	GRU, LSTM, RUSS
Skachkova et al. (2018)	SRNN, GRU, LSTM
Sennhauser and Berwick (2018)	LSTM
Suzgun et al. (2019)	SRNN, GRU, LSTM
Yu et al. (2019)	seq2seq

Table 4: Overview of investigated models.

While all these papers use a formal language to evaluate models, several factors prevent them from forming a solid basis upon which to compare their respective results: First, there is neither a benchmark train/dev/test set for Dyck languages (as is standard for most machine learning tasks) (Table 2), nor a set of measures that is reported consistently throughout the literature (Table 3). Additionally, only two papers compare the three well-established architectures (SRNN, GRU, LSTM) directly (Table 4). Finally, the employed training and test measures are not unified - in Bernardy (2018), for example, the models are trained to predict the next letter in words of variable length at any given time step, while in Suzgun et al. (2019), they predict the final letter of a word. Further model details, such as the inclusion of an Embedding and/or a Dropout layer or the number of hidden units also vary.

In conclusion, despite formal languages having been used to assess neural network model performance for decades, there is little to no comparative studies of SRNN, LSTM and GRU models performing on D_2. Any research comparing new architectures to any of these models does so with both varying training and testing methods, as well as with vastly differing corpora and as such cannot be directly compared with each other. This allows for no conclusive statement on the relative performance of these popular RNN architectures based on the current literature.

3 Experiment Setup

3.1 Evaluation

Generating the words for the two experiments follows the procedure described by Bernardy (2018) and will be explained in depth in the coming sections. Whereas the research put forth in his paper scrutinized the generative abilities of RNNs, I am investigating the models performing a classification task. As such, my training, test, validation and experiment data all consist of the same 1:1 ratio of correct-to-incorrect words. Within the incorrect words, a distinction between superfluous opening or closing brackets is made, also at a ratio of 1:1.

As such, a random guessing strategy would yield a baseline accuracy of 50%. A model is considered as having learned useful features from the training data if it scores above the baseline accuracy in the experiments. Furthermore, if the model has learned the underlying grammar of D_2, there should be no difference in accuracy for the two classes of incorrect words, as both of them do not belong to D_2, regardless of which bracket is replaced.

3.2 Models

For the following experiments, the three RNN architectures described in Section 2.4 have been used. All models consist of an embedding layer, a single layer of size $n = \{2^1, 2^2, \ldots, 2^n\}$ and a dense layer of size 1 with sigmoid activation. The activation value of the neuron in the dense layer acts as the output of the model: a value ≥ 0.5 means the model classified the input as a correct word. The models were implemented in Tensorflow 2.0. All code was implemented using Python `3.7.6`, NumPy `1.18.1` (McKinney, 2011), pandas `0.25.3` (Oliphant, 2006) and scikit-learn `0.22.1` (Pedregosa et al., 2011).[4]

All models were trained with the same parameters. The training data was received one word at a time, in batches of 512. The loss was computed by binary cross-entropy, as is current standard for binary classification tasks. Furthermore, the Adam optimizer (Kingma and Ba (2014)) was applied with a learning rate of 0.0001. At the end of a training epoch, the models were evaluated for loss and accuracy on a validation set of 120,000 words. The models were trained until their loss on the validation set did not lower by more than 0.0001 for three consecutive epochs or for at most 100 epochs. The models with the lowest validation loss were used for all experiments.

The models were trained on the same training data for both experiments. To answer the question of training data influence on model performance, three distinct sets of training data were used, yielding a total of $9 \times 3 \times 3 = 81$ (number of different hidden

[4]The full source code can be found in the Appendix or under `https://github.com/FyDob/BSc-Thesis`.

Corpus	*Word Length*	*maxND*	*maxBD*
Baseline	18.37 (6.36)	4.31 (1.22)	13.00 (16.02)
High LRD	18.67 (4.75)	5.12 (1.04)	16.67 (4.75)
Low LRD	17.54 (8.04)	3.92 (0.98)	10.58 (9.02)

Table 5: Properties of the three corpora the models were trained on, reported in averages (variance in brackets).

units $\times$ number of training corpora $\times$ number of different architectures) evaluated models.

3.3 Corpus Construction

To investigate the influence of corpus composition on model performance, three corpora were created: a baseline corpus which is directly sampled from a subset of D_2, as well as two modifications of the baseline corpus: one impoverishing the training data from long-range dependencies (Low LRD) and one enriching the training data with more long-range dependencies (High LRD). The sampling and modification processes will be explained later in this section.

The experiments were explicitly designed to test the models' abilities to generalize based on the training data they encounter. As such, it is prudent to give consideration to which properties the training data might possess to facilitate or inhibit generalizability - properties such as length, maximum nesting depth (ND) and the maximum distance between a pair of opening and closing brackets (BD). ND is, in this case, defined as the highest number of unresolved open brackets preceding an open bracket in a given word (i.e. in the word `{[{}]}`, the square open bracket is at ND$=1$, and the curly open bracket is at ND$=2$, making the maximum ND of the word 2). Maximum BD, then, is the highest number of characters between a pair of brackets in a word. In the previous example word, the maximum BD would be 4. These measures are reported in Table 5 in terms of averages and variance.

Furthermore, the training corpora were chosen to be a small slice of a comparatively large subset of D_2. To facilitate generalization, the training corpora consist of words of varying length. As discussed in Section 2.5, previous works largely utilized similarly small language subsets and achieved encouraging results. For a discussion of Experiment 1 and 2 on a training corpus consisting of a majority of the target language, see Bernardy (2018).

In determining an eligible maximum length, a known fact about the size of D_n subsets was utilized: a Dyck language D_n contains $n^m C_m$ words of length $2m$, where C_m is the m-th Catalan number (Skachkova et al. (2018)). It follows that a maximum length limit of $2m$ produces a set of size $\sum_{i=2}^{2m} n^i C_i$. For example, a maximum length of 20 in D_2 ($D_2^{\leq 20}$) yields 20,119,506 words, which is a sufficiently large subset to sample from. The words were generated following the probabilistic grammar set forth by Sennhauser and Berwick (2018).

$$
\begin{aligned}
&S \rightarrow Z\ S \mid Z \\
&Z \rightarrow B \mid T \\
&B \rightarrow [\ S\ ] \mid \{\ S\ \} \\
&T \rightarrow [\ ] \mid \{\ \}
\end{aligned}
$$

The production $Z \rightarrow B$ branches, whereas $S \rightarrow Z\ S$ concatenates two smaller Dyck words. This representation provides a good intuition for understanding the merit of Experiment 1. The probabilities with which the rules were applied are calculated as follows, with alternative rules of course being applied with the complementary probability:

$$
\begin{aligned}
P_{\text{branch}} &= r_{\text{branch}} \cdot s(l) \quad \text{with } r_{\text{branch}} \sim U(0.7, 1.0) \\
P_{\text{concat}} &= r_{\text{concat}} \cdot s(l) \quad \text{with } r_{\text{concat}} \sim U(0.7, 1.0) \\
s(l) &= \min(1, -3 \cdot \frac{l}{n} + 3)
\end{aligned}
$$

with l being the number of already generated non-terminal characters and n the maximally desired length of the word. r_{branch}, r_{concat} and l were sampled at every step of word generation.

Following this process, 500,000 words in $D_2^{\leq 20}$ were generated. These words served as the basis for creating the three corpora. To create the Low LRD corpus, all words with a maximum bracket distance higher than 10 were modified[5] by first identifying the bracket pair with the highest bracket distance, then simply moving the opening bracket from its original position to the position right before the closing bracket. (i.e. `{[{}]}` becomes `[{}]{}`). This has the largest impact on bracket distance throughout the corpus, while ensuring grammaticality of the resulting word. The resulting set of long-range impoverished words was merged with all unmodified words, deleting all duplicates.

The High LRD corpus was created in a similar way: First, all words with a bracket distance lower than 19 were identified.[6] Then, the first pair of neighbouring closing brackets is found and deleted. The remaining word is wrapped in a randomly chosen pair of brackets, creating the longest possible bracket distance between the two (i.e. `{[{}[]]}` becomes `{{[[]]}}`). The resulting set was merged with the unmodified words the same way as the Low LRD set.

Finally, the corpora were filled with 500,000 non-words obtained by corrupting the

[5]This cut-off point was chosen as it significantly reduces the average maximum bracket distance without creating too many duplicates.

[6]The same considerations as for the Low LRD corpus cut-off apply.

correct words in $D_2^{\leq 20}$. For one half of the words, a random opening bracket was replaced with a random closing bracket, while a random closing bracket was replaced with a random opening bracket for the other half.

In total, all corpora consist of 1,000,000 samples, of which 50% are incorrect.

3.4 Experiment 1: Long-Range Dependency

For this experiment, the test set consisted of 1,000,000 samples of length $1+18+18+1 = 38$, half of which were correct Dyck words. They were created by picking two random Dyck words $w_1, w_2 \in D_2^{=18}$ from the base corpus, concatenating them and wrapping the result in a randomly selected pair of matching brackets as follows:

$$w_{\text{LRD}} = O_n w_1 w_2 C_n$$

To generate incorrect samples, the generated correct LRD words were corrupted in the same way as for the training corpora, yielding 250,000 incorrect LRD words with a superfluous opening or closing bracket each.

While w_1 and w_2 might have been seen in training (for models trained on the base corpus), the resulting word most certainly has not been observed. Neither could the model possibly have encountered a long-range dependency spanning 36 characters between the opening and closing bracket. As such, a high classification accuracy serves as a strong indication of the model having learned to generalize to longer, non-concatenated Dyck words.[7] I report model performance on Experiment 1 in terms of accuracy, precision, recall and F1 score.

3.5 Experiment 2: New Depths

To investigate how well a model performs on predicting brackets on a nesting level deeper than anything included in training, another test set was constructed. Since Experiment 1 already investigates Long-Range Dependency (LRD), this corpus was designed so its results are confounded as little as possible by LRD performance.

For this task, the test set consisted of 1,000,000 samples of length 30, half of which were correct Dyck words. First, $500,000$ correct words were chosen at random from the base corpus. Then, they were wrapped by a prefix of five randomly chosen opening brackets and a suffix of the corresponding closing brackets as follows:

$$w_{\text{DN}} = O_n O_n O_n O_n O_n w C_n C_n C_n C_n C_n$$

Generation of incorrect samples was done in accordance to Experiment 1 and corpus

[7]While the infixed sub-words are indeed concatenated, w_{LRD} cannot be created by concatenating two shorter words due to being wrapped by a matching bracket pair.

creation.

This process still has the model extrapolate beyond the length of the training words, while increasing all present nesting depths by 5. This is analogous to center embedding in natural language - processing increasing nesting levels is more complicated than processing a flat structure. A high classification accuracy in Experiment 2 indicates a capability to generalize to repeated application of grammar rules beyond what was seen in the training set. As such, it implies an understanding of the D_2 grammar. I report model performance on Experiment 2 in terms of accuracy, precision, recall and F1 score.

4 Results

During training, almost all 81 trained models achieved a validation accuracy near 100%, except for the SRNN-2 models trained on the base and low LRD corpus, which scored 75.0% and 50.4% respectively. I have included them in the experiments regardless of their low validation accuracy, since it was unclear whether validation accuracy would be a strong predictor for a network's performance on the experiment data. I present my results with regard to three focus points: First, the overall performance of different architectures with respect to which corpus they were trained on, then the individual model performances on each of the two experiments, and finally a closer look at classifications made by outlier networks - networks which drastically over- or underperformed in either of the experiments - with regards to misclassified false positives.

4.1 Architecture/Training Data

As can be seen in Table 6, none of the architectures consistently achieved an accuracy far above the random guessing baseline of 50.0%. However, there was still a notable difference in performance between architectures: on average, the GRU networks scored the highest on accuracy and precision, while the SRNN networks achieved the best recall and F1 score. With 51.5%, LSTMs scored an average accuracy between SRNNs (50.0%) and GRUs (53.3%), but they underperformed in all other experiment measures.

Furthermore, the choice of training data had a notable effect on overall model performance: SRNNs and GRUs received a boost in performance in all measures when comparing the Base to the Low LRD models, elevating SRNNs from an accuracy below random guessing to 51.4%. While LSTMs lost 1.4% in terms of accuracy, all other performance measures improved significantly for the Low LRD models. Training on the High LRD corpus aided SRNNs in terms of accuracy, precision and F1 score, but worsened accuracy, recall and F1 score for LSTMs and GRUs.

4.2 Experiment 1: Long-Range Dependency

I report results for Experiment 1 in Tables 7, 8 and 9, which include the performance measures for all networks trained on the Base, Low and High LRD corpus respectively, as evaluated on Experiment 1.

When trained on the Base corpus, 8 of 27 models (29.6%) achieved an accuracy above random guessing. Among those 8, only 2 reached an accuracy above 60%: LSTM-8 and GRU-2. The vast majority of models - 20 in total - reached an accuracy of $50 \pm 5\%$. 4 performed even worse than that: the worst model (SRNN-16) only achieved 27.2% accuracy on the experiment data. There was no apparent correlation between validation accuracy and model performance in Experiment 1 - indeed, SRNN-2 with the lowest validation accuracy at 75.0% evaluated at below chance, but so did several models with a validation accuracy of 100%.

12 of 27 Low LRD-trained models (44.4%) scored an accuracy higher than the random guessing baseline. Among those, 3 models - SRNN-4, LSTM-16 and GRU-64 - reached an accuracy above 60%. While 22 models fell within the $50 \pm 5\%$ belt of accuracy, only 1 model performed significantly worse: LSTM-4 with 27.8%. Validation accuracy was entirely unrelated to model performance, with LSTM-4 having achieved a perfect score on the validation data, but completely failing at Experiment 1. On average, all measures have improved when compared to the Base models: accuracy improved by +2.7 percentage points (p.p.), precision by +11.0 p.p., recall by +9.5 p.p. and F1 score by +8.3 p.p.

9 of 27 High LRD-trained models (33.3%) achieved an accuracy above baseline, but only GRU-512 by a significant margin with 80%. 23 models performed within the $\pm 5\%$ margin around the baseline, and 3 models (LSTM-4, LSTM-8 and GRU-8) underperfomed significantly. There was no relation between validation accuracy and performance on experiment data for High LRD models, either. Compared to the Base models, High LRD models almost consistently scored worse: accuracy, recall and F1 score went down by -1.0, -2.2 and -1.8 p.p. respectively, but precision was improved by +5.6 p.p.

4.3 Experiment 2: New Depths

All results for Experiment 2 can be found in Tables 10, 11 and 12. When comparing mean performances across all models, they largely scored higher on Experiment 2 than Experiment 1. This suggests that Experiment 2 was easier regardless of training data. As with Experiment 1, a model's validation accuracy did not correlate with its performance on the experiment data.

Among the Base models, 8 of 27 (29.6%) performed above guessing baseline in Experiment 2 - the same ratio as for Experiment 1, though there was minimal overlap in the best performers. Only GRU-64 and SRNN-64 performed above 50% accuracy for both experiments. 20 of 27 models stayed in the ±5% margin of the baseline, with only 2 (SRNN-128 and SRNN-256) dropping below that. The best performing network - LSTM-128 - scored the highest accuracy across all models and all experiments with 99.3%.

The same number of Low LRD models performed above chance for Experiment 2 as for Experiment 1 (44.4%), with 6 models (SRNN-2, SRNN-8, LSTM-16, LSTM-512, GRU-4 and GRU-64) occuring in both groups. LSTM-512 and GRU-64 have both achieved an accuracy $> 90\%$. 19 models performed within the ±5% margin around the baseline, only 2 dropped below that. Compared to the Base models, low LRD models also performed better on Experiment 2: mean accuracy is up by +0.4 p.p., precision by +20.6, recall by +3.5 and F1 score by 6.5 p.p.

While the highest number of High LRD models have achieved an accuracy above random guessing - 10 of 27, or 37.0% - only SRNN-16 crossed the 60% threshold at all. Indeed, aside from SRNN-16, only one other model lay outside of the ±5% margin around the baseline - LSTM-64, with an accuracy of 28.5%. When comparing with the Base models, all measures except for precision, which improved by +10.7 p.p. Accuracy went down by -2.7 p.p., recall by -7.5 and F1 score by -5.0 p.p.

4.4 Outliers: A Closer Look

As is evident from the results described so far, a vast majority of models fell within a ±5% margin around random guessing in terms of accuracy. I consider these models to not have extracted any useful grammar information from the training data and discard them for further investigation. As such, I will only discuss models with an accuracy either $> 55\%$ or $< 45\%$, to both gather information on what caused models to succeed, and what caused them to fail.

	Accuracy	*Precision*	*Recall*	*F1 Score*	*Val Acc*
SRNN			Base		
Mean	*0.476*	*0.278*	*0.181*	*0.193*	0.972
Variance	0.064	0.235	0.257	0.220	0.081
SRNN			Low LRD		
Mean	**0.514**	**0.443**	**0.288**	**0.316**	*0.932*
Variance	0.055	0.176	0.259	0.213	0.158
SRNN			High LRD		
Mean	0.511	0.415	0.153	0.200	**0.981**
Variance	0.043	0.211	0.171	0.187	0.054
SRNN			Complete		
Mean	*0.500*	0.379	**0.208**	**0.236**	*0.962*
Variance	0.056	0.217	0.236	0.211	0.107
LSTM			Base		
Mean	**0.543**	*0.219*	0.148	0.154	0.999
Variance	0.173	0.365	0.335	0.341	0.003
LSTM			Low LRD		
Mean	0.529	**0.391**	**0.176**	**0.195**	0.999
Variance	0.158	0.295	0.311	0.302	0.002
LSTM			High LRD		
Mean	*0.472*	0.258	*0.036*	*0.059*	**1.000**
Variance	0.075	0.278	0.057	0.090	0.000
LSTM			Complete		
Mean	0.515	*0.289*	*0.120*	*0.136*	**0.999**
Variance	0.143	0.318	0.268	0.269	0.002
GRU			Base		
Mean	0.531	*0.371*	0.147	0.185	0.999
Variance	0.097	0.297	0.226	0.247	0.001
GRU			Low LRD		
Mean	**0.554**	**0.507**	**0.206**	**0.243**	0.999
Variance	0.133	0.198	0.302	0.285	0.001
GRU			High LRD		
Mean	*0.514*	0.439	*0.126*	*0.170*	*0.972*
Variance	0.082	0.223	0.198	0.193	0.055
GRU			Complete		
Mean	**0.533**	**0.439**	0.160	0.200	0.990
Variance	0.106	0.245	0.244	0.242	0.034

Table 6: Performance measures of all architectures across both experiments depending on which corpus they were trained on, as well as the compounded measures for all networks regardless of training data.

Network	*Accuracy*	*Precision*	*Recall*	*F1 Score*	*Val Acc*
GRU-2	0.890	**0.982**	0.794	0.878	0.995
GRU-4	0.487	0.413	0.060	0.105	1.000
GRU-8	0.488	0.331	0.023	0.043	1.000
GRU-16	0.550	0.713	0.168	0.272	1.000
GRU-32	0.434	0.267	0.075	0.118	1.000
GRU-64	0.510	0.537	0.149	0.234	1.000
GRU-128	0.553	0.611	0.293	0.396	0.999
GRU-256	0.497	0.364	0.009	0.018	1.000
GRU-512	0.500	0.487	0.007	0.015	1.000
LSTM-2	0.500	0.000	0.000	0.000	0.999
LSTM-4	0.451	0.000	0.000	0.000	1.000
LSTM-8	**0.910**	0.959	**0.857**	**0.905**	**1.000**
LSTM-16	0.343	0.001	0.000	0.000	1.000
LSTM-32	0.500	0.000	0.000	0.000	1.000
LSTM-64	0.505	0.596	0.030	0.057	1.000
LSTM-128	0.347	0.001	0.000	0.000	1.000
LSTM-256	0.455	0.128	0.016	0.028	1.000
LSTM-512	0.499	0.468	0.009	0.017	0.991
SRNN-2	0.465	0.357	0.087	0.140	*0.750*
SRNN-4	0.488	0.334	0.023	0.043	1.000
SRNN-8	0.461	0.037	0.003	0.006	1.000
SRNN-16	*0.272*	0.021	0.010	0.014	1.000
SRNN-32	0.498	0.492	0.141	0.219	1.000
SRNN-64	0.504	0.505	0.366	0.424	1.000
SRNN-128	0.484	0.040	0.001	0.003	1.000
SRNN-256	0.503	0.503	0.455	0.478	1.000
SRNN-512	0.484	0.017	0.001	0.001	1.000
Mean	0.503	0.339	0.132	0.163	0.990
Variance	0.130	0.294	0.233	0.253	0.048

Table 7: Performance measures for Experiment 1 of all networks that were trained on the Base LRD corpus.

Network	*Accuracy*	*Precision*	*Recall*	*F1 Score*	*Val Acc*
GRU-2	0.583	0.582	0.589	0.585	1.000
GRU-4	0.505	0.518	0.135	0.214	1.000
GRU-8	0.498	0.491	0.106	0.175	0.997
GRU-16	0.503	0.528	0.060	0.107	1.000
GRU-32	0.505	0.525	0.100	0.168	1.000
GRU-64	**0.890**	**0.869**	**0.918**	**0.893**	**1.000**
GRU-128	0.484	0.324	0.030	0.054	1.000
GRU-256	0.500	0.496	0.039	0.072	1.000
GRU-512	0.500	0.435	0.002	0.004	0.997
LSTM-2	0.500	0.500	0.475	0.487	0.996
LSTM-4	*0.278*	*0.000*	*0.000*	*0.000*	*1.000*
LSTM-8	0.500	0.000	0.000	0.000	1.000
LSTM-16	0.881	0.853	0.920	0.885	1.000
LSTM-32	0.491	0.092	0.002	0.004	1.000
LSTM-64	0.500	0.505	0.010	0.020	1.000
LSTM-128	0.496	0.374	0.011	0.021	0.998
LSTM-256	0.500	0.535	0.007	0.014	1.000
LSTM-512	0.507	0.566	0.058	0.106	1.000
SRNN-2	0.501	0.501	0.366	0.423	0.*504*
SRNN-4	0.708	0.646	0.922	0.760	0.930
SRNN-8	0.508	0.527	0.150	0.234	1.000
SRNN-16	0.500	0.000	0.000	0.000	1.000
SRNN-32	0.486	0.388	0.050	0.088	1.000
SRNN-64	0.487	0.375	0.040	0.072	1.000
SRNN-128	0.492	0.484	0.236	0.318	0.953
SRNN-256	0.501	0.501	0.412	0.452	1.000
SRNN-512	0.503	0.503	0.497	0.500	1.000
Mean	0.530	0.449	0.227	0.246	0.977
Variance	0.120	0.217	0.304	0.280	0.096

Table 8: Performance measures for Experiment 1 of all networks that were trained on the Low LRD corpus.

Network	*Accuracy*	*Precision*	*Recall*	*F1 Score*	*Val Acc*
GRU-2	0.494	0.480	0.136	0.212	1.000
GRU-4	0.499	0.482	0.036	0.066	1.000
GRU-8	0.342	0.000	0.000	0.000	1.000
GRU-16	0.523	0.590	0.153	0.242	1.000
GRU-32	0.504	0.521	0.093	0.158	1.000
GRU-64	0.504	0.522	0.088	0.150	1.000
GRU-128	0.489	0.419	0.056	0.098	1.000
GRU-256	0.503	0.549	0.035	0.066	0.902
GRU-512	**0.800**	**0.768**	**0.861**	**0.812**	0.849
LSTM-2	0.483	0.390	0.060	0.104	1.000
LSTM-4	*0.282*	*0.000*	*0.000*	*0.000*	1.000
LSTM-8	0.379	0.000	0.000	0.000	1.000
LSTM-16	0.502	0.640	0.007	0.015	1.000
LSTM-32	0.498	0.475	0.038	0.070	1.000
LSTM-64	0.500	0.000	0.000	0.000	1.000
LSTM-128	0.492	0.365	0.022	0.041	1.000
LSTM-256	0.497	0.367	0.008	0.016	1.000
LSTM-512	0.500	0.000	0.000	0.000	0.999
SRNN-2	0.500	0.499	0.042	0.078	*0.833*
SRNN-4	0.502	0.523	0.046	0.085	0.996
SRNN-8	0.539	0.622	0.201	0.304	1.000
SRNN-16	0.500	0.000	0.000	0.000	0.999
SRNN-32	0.500	0.497	0.037	0.070	1.000
SRNN-64	0.503	0.504	0.360	0.420	1.000
SRNN-128	0.497	0.494	0.235	0.318	1.000
SRNN-256	0.496	0.495	0.340	0.403	1.000
SRNN-512	0.490	0.459	0.112	0.180	1.000
Mean	0.493	0.395	0.110	0.145	0.984
Variance	0.083	0.230	0.180	0.182	0.045

Table 9: Performance measures for Experiment 1 of all networks that were trained on the High LRD corpus.

Network	*Accuracy*	*Precision*	*Recall*	*F1 Score*	*Val Acc*
GRU-2	0.500	0.000	0.000	0.000	0.995
GRU-4	0.563	0.655	0.265	0.377	1.000
GRU-8	0.472	0.164	0.014	0.025	1.000
GRU-16	0.500	0.000	0.000	0.000	1.000
GRU-32	0.500	0.000	0.000	0.000	1.000
GRU-64	0.515	0.550	0.160	0.248	1.000
GRU-128	0.496	0.000	0.000	0.000	0.999
GRU-256	0.500	0.000	0.000	0.000	1.000
GRU-512	0.601	0.597	0.620	0.608	1.000
LSTM-2	0.500	0.000	0.000	0.000	0.999
LSTM-4	0.500	0.000	0.000	0.000	1.000
LSTM-8	0.500	0.000	0.000	0.000	1.000
LSTM-16	0.500	0.000	0.000	0.000	1.000
LSTM-32	0.778	0.792	0.755	0.773	1.000
LSTM-64	0.500	0.000	0.000	0.000	1.000
LSTM-128	**0.993**	**0.991**	**0.995**	**0.993**	**1.000**
LSTM-256	0.500	0.000	0.000	0.000	1.000
LSTM-512	0.499	0.000	0.000	0.000	0.991
SRNN-2	0.580	0.548	0.917	0.686	*0.750*
SRNN-4	0.488	0.000	0.000	0.000	1.000
SRNN-8	0.500	0.000	0.000	0.000	1.000
SRNN-16	0.500	0.000	0.000	0.000	1.000
SRNN-32	0.515	0.564	0.136	0.220	1.000
SRNN-64	0.511	0.529	0.192	0.281	1.000
SRNN-128	0.441	0.409	0.266	0.322	1.000
SRNN-256	*0.382*	0.156	0.053	0.080	1.000
SRNN-512	0.497	0.497	0.615	0.550	1.000
Mean	0.531	0.239	0.185	0.191	0.990
Variance	0.113	0.311	0.308	0.290	0.048

Table 10: Performance measures for Experiment 2 of all networks that were trained on the Base LRD corpus.

Network	*Accuracy*	*Precision*	*Recall*	*F1 Score*	*Val Acc*
GRU-2	0.494	0.488	0.222	0.306	1.000
GRU-4	0.561	0.651	0.262	0.374	1.000
GRU-8	0.499	0.492	0.056	0.101	0.997
GRU-16	0.497	0.465	0.039	0.072	1.000
GRU-32	0.500	0.000	0.000	0.000	1.000
GRU-64	0.935	0.910	0.966	0.937	1.000
GRU-128	0.516	0.560	0.144	0.229	1.000
GRU-256	0.499	0.486	0.043	0.078	1.000
GRU-512	0.497	0.305	0.004	0.008	0.997
LSTM-2	0.499	0.000	0.000	0.000	0.996
LSTM-4	0.480	0.328	0.037	0.067	1.000
LSTM-8	0.555	0.660	0.226	0.337	1.000
LSTM-16	0.566	0.662	0.270	0.384	1.000
LSTM-32	*0.327*	0.207	0.122	0.154	1.000
LSTM-64	0.491	0.301	0.014	0.026	1.000
LSTM-128	0.500	0.000	0.000	0.000	0.998
LSTM-256	0.502	0.538	0.027	0.051	1.000
LSTM-512	**0.953**	**0.917**	**0.996**	**0.955**	**1.000**
SRNN-2	0.501	0.501	0.585	0.540	*0.504*
SRNN-4	0.442	0.361	0.149	0.211	0.930
SRNN-8	0.529	0.582	0.206	0.304	1.000
SRNN-16	0.498	0.000	0.000	0.000	1.000
SRNN-32	0.580	0.561	0.738	0.637	1.000
SRNN-64	0.514	0.546	0.170	0.260	1.000
SRNN-128	0.498	0.495	0.178	0.262	0.953
SRNN-256	0.500	0.499	0.348	0.410	1.000
SRNN-512	0.501	0.505	0.136	0.214	1.000
Mean	0.535	0.445	0.220	0.256	0.977
Variance	0.126	0.246	0.281	0.263	0.096

Table 11: Performance measures for Experiment 2 of all networks that were trained on the Low LRD corpus.

Network	*Accuracy*	*Precision*	*Recall*	*F1 Score*	*Val Acc*
GRU-2	0.535	0.580	0.253	0.353	1.000
GRU-4	0.502	0.518	0.070	0.123	1.000
GRU-8	0.500	0.000	0.000	0.000	1.000
GRU-16	0.492	0.269	0.010	0.019	1.000
GRU-32	0.511	0.552	0.114	0.189	1.000
GRU-64	0.502	0.512	0.081	0.140	1.000
GRU-128	0.541	0.601	0.244	0.347	1.000
GRU-256	0.503	0.530	0.048	0.089	0.902
GRU-512	0.500	0.000	0.000	0.000	0.849
LSTM-2	0.531	0.591	0.200	0.299	1.000
LSTM-4	0.500	0.000	0.000	0.000	1.000
LSTM-8	0.500	0.000	0.000	0.000	1.000
LSTM-16	0.517	0.658	0.070	0.127	1.000
LSTM-32	0.511	0.554	0.112	0.187	1.000
LSTM-64	*0.285*	0.000	0.000	0.000	1.000
LSTM-128	0.500	0.000	0.000	0.000	1.000
LSTM-256	0.521	0.608	0.121	0.202	1.000
LSTM-512	0.500	0.000	0.000	0.000	0.999
SRNN-2	0.491	0.333	0.017	0.033	*0.833*
SRNN-4	0.500	0.000	0.000	0.000	0.996
SRNN-8	0.500	0.000	0.000	0.000	1.000
SRNN-16	**0.675**	**0.691**	**0.633**	**0.661**	0.999
SRNN-32	0.507	0.557	0.072	0.128	1.000
SRNN-64	0.492	0.473	0.139	0.215	1.000
SRNN-128	0.485	0.281	0.019	0.036	1.000
SRNN-256	0.501	0.502	0.322	0.392	1.000
SRNN-512	0.512	0.534	0.183	0.273	1.000
Mean	0.504	0.346	0.100	0.141	0.984
Variance	0.056	0.267	0.141	0.164	0.045

Table 12: Performance measures for Experiment 2 of all networks that were trained on the High LRD corpus.

Architecture	*Experiment*	*Training Corpus*	*Open*	*Closed*	*Ratio*	*Total*
			High Accuracy Models			
LSTM	LRD	base	18,161	0	inf	18,161
GRU	LRD	base	55,256	45,207	1.222	100,463
SRNN	LRD	low	101,557	150,905	0.673	252,462
LSTM	LRD	low	39,584	39,559	1.001	79,143
GRU	LRD	low	165,820	115,236	1.439	281,056
GRU	LRD	high	43,497	86,753	0.501	130,250
SRNN	ND	base	138,673	240,098	0.578	378,771
LSTM	ND	base	103,816	0	inf	103,816
GRU	ND	base	244,265	34,843	7.010	279,108
SRNN	ND	low	94,602	194,386	0.487	288,988
LSTM	ND	low	97,527	74,625	1.307	172,152
GRU	ND	low	82,933	34,981	2.371	117,914
SRNN	ND	high	90,850	50,458	1.801	141,308
			Low Accuracy Models			
SRNN	LRD	base	228,451	4,387	52.075	232,838
LSTM	LRD	base	298,547	11,358	26.285	309,905
GRU	LRD	base	29,908	73,284	0.408	103,192
LSTM	LRD	low	222,045	0	inf	222,045
LSTM	LRD	high	338,852	387	875.587	339,239
GRU	LRD	high	0	157,850	0.000	157,850
SRNN	ND	base	307,497	29,619	10.382	337,116
SRNN	ND	low	116,613	15,213	7.665	131,826
LSTM	ND	low	234,224	0	inf	234,224
LSTM	ND	high	214,980	0	inf	214,980

Table 13: Ratio of open/closed error categories misclassified as false positives.

First, I report the number of the two error categories (superfluous open bracket, superfluous closed bracket) per network architecture in Tables 13 and. Hypothetically, since both categories were balanced in training and experiment data, there should be no significant difference between misclassifying one or the other - unless one category proves to be more complex to an architecture. The ratio of open-to-closed bracket misclassifications serves as a simple indication of whether the model extracted the correct information - that the amount of brackets needs to be balanced throughout a valid word - at all. A ratio of 1 implies no model difference between the two categories. Conversely, any skewing above or below 1 shows the model disproportionally struggling with one of the two categories. Only one model achieved a near perfect ratio: LSTM-16 trained on the Low LRD corpus, evaluating the Experiment 1 data.

Wrong words with superfluous open brackets have shown to be the most difficult to reject: 61.54% of the High Accuracy Models and 80% of the Low Accuracy Models skew towards a ratio > 1. Furthermore, a model's inability to treat both error categories the same as indicated by the ratio correlates to model accuracy: will most High Accuracy Models err reasonably closely to 1, the Low Accuracy Models show a much more extreme distribution, frequently misclassifying almost all words in one error category, while handling the other one perfectly. There is no trend towards one architecture overall skewing towards 1. However, among the High Accuracy Models, the models trained on the Low LRD corpus have the lowest deviation from 1, with 0.256 on Experiment 1, and 0.652 on Experiment 2. The lowest deviation from 1 among the Low Accuracy Models comes from the models trained on the Base corpus, with 25.65 and 9.382, respectively.

5 Discussion

The results presented in Chapter 4 provide an in-depth look at model performance, reporting on established measures like accuracy and F1 score, but also on custom measures for this task, like misclassification ratio for the two categories of incorrect words. All of this was done to answer the question of whether RNNs can learn the underlying structure of D_2, and whether specific properties of the training data can facilitate or inhibit that ability. I will now interpret the results from the perspective of these questions.

5.1 Learning D_2

Both experiments were designed to test the two most important aspects of what it means to generalize from a small language subset to more complex data: interpreting extreme long-range dependencies on long, unseen words, and handling unseen nesting depths. The former showcases the ability of generalizing to much greater length, while the latter provides insight on how well the model handles deeper, more complex parse trees.

Judging from the average accuracy across all models reported in Tables 7, 8, 9, 10, 11 and 12, Experiment 1 was a harder task than Experiment 2, regardless of training corpus. By and large, models were struggling to correctly classify extreme long-range dependencies, while performing well on a deeper nesting task with shorter dependencies. Assuming that nesting depth, as it creates deeper and more complicated underlying parse trees than low-nesting depth long-range dependencies, would pose a higher difficulty to models that have learned to build a structural representation and as such, learned D_2, this discrepancy suggests that most models did not achieve such a representation.

In conclusion, the experiments posed in this thesis were hard tasks. Not many models learned helpful information from the training data, some extracted the wrong kind of information, but most learned nothing to help them improve from random guessing. Successful model hidden unit counts range from 2 to 512, with no general trend connecting number of hidden units and model performance, making it impossible to assess lower and upper limits of model complexity required to learn D_2.

However, 20 models in total have performed above chance on either of the two experiments. Accuracy ranges from 80.0% (High GRU-512) to 91.0% (Base LSTM-8) for Experiment 1, and 67.5% (High SRNN-16) to 99.3% (Base LSTM-128) for Experiment 2. Regardless of experiment and training data, GRUs slightly outperformed LSTMs and SRNNs with an average accuracy of 53.5% (see Table 6).

Individual model performance provides another intriguing implication: Several models that performed well in Experiment 1 barely achieved the baseline in Experiment 2 and vice-versa. This points to the tasks, despite both making a case for generalizability, posing different requirements to the representations the models learned. In fact, only two models performed above baseline for both experiments: Low LSTM-16 (88.1% and 56.66% for Experiment 1 and 2, respectively) and Low GRU-64 (89.0% and 93.5%). Considering there were 20 successful models in total, this number is fairly low, further pointing to learning D_2 being a difficult undertaking. Both of these models have been trained on the Low LRD corpus, indicating that the corpus facilitates all-purpose generalization.

To assess whether a High Accuracy model learned a valid representation of D_2, incorrect words were split in two equal sized classes: words with superfluous open or closed brackets. A good model, then, should not make a difference between the two classes. While the High Accuracy models did not always succeed at that, they came a lot closer to treating both classes equally than the Low Accuracy models. Furthermore, the imbalance in false positive misclassification was skewed towards words with superfluous open brackets. This might be owed to the fact that an extra closed bracket at any position immediately renders a word of any length ungrammatical: it resolves a dependency that does not exist. An extra open bracket, however, opens a dependency that might be resolved at a later point in the word. Indeed, all incorrect open bracket words are substrings of longer correct D_2 words. The same cannot be said for incorrect closed bracket words.[8]

In conclusion, the combination of task difficulty and volatile model performance makes it difficult to conclusively compare the three architectures. While GRUs achieved a high accuracy more consistently, LSTMs have produced the highest scoring models. Despite SRNNs scoring the worst in terms of overall accuracy, they - on average - outperformed the other two architectures in terms of F1 score. However, given the measure of misclassification ratio, the record high accuracy and fairly low number of hidden units in well-performing models, LSTMs have shown great promise to be capable of learning D_2 in this study.

5.2 Influence of Training Data

Whether comparing accuracy, precision, recall and F1 score across both experiments (see Table 6, individually comparing model instances based on what corpus they were trained on (i.e. Tables 7, 8 and 9 for Experiment 1) or looking for a trend in open/-closed misclassification ratio (see Table 13): the effect of training corpus complexity is consistent. Disregarding experiments, SRNN and GRU performance improved in all measures when trained on the Low LRD corpus. For LSTMs, it improved every measure but accuracy. When distinguishing between both experiments, models trained on the Low LRD corpus performed the best on average. Furthermore, High Accuracy Models showed the lowest deviation from the ideal 1 to 1 open/closed misclassification ratio if they were trained on the Low LRD corpus. Finally, Low LRD trained models constitute the majority class of High Accuracy Models. On the other hand, models trained on the High LRD corpus on average underperformed compared to the Base models, regardless of model architecture and experiment.

Overall, there is strong evidence for the Low LRD training corpus leading to the most consistently good results, while the High LRD corpus tends to worsen models. This is surprising: considering how the High LRD corpus is constructed, a sizeable portion of training data is similar to the experiment data, featuring at least one long-range dependency spanning the whole word. Frequently encountering this pattern in training as well as having the model encounter more complex structures seemed likely to boost model performance. Instead, the opposite is true: learning from a less complex corpus enhanced robustness and the ability to generalize, while complex training data inhibited these processes.

[8]It must be noted that this property makes the incorrect open words in no way more valid than the incorrect closed words. The experimental stimuli are of a fixed length, and their end is - like in the training data - signified by an end-of-word symbol. There is no reason for a model to anticipate an extension to resolve the open dependency.

These results imply not only that RNNs generalize from limited data to longer, more complex examples, they also do so by extracting generative rules from an underlying structure. This process is facilitated by giving the RNNs simple examples: once the connection is made that an open bracket must eventually be closed by its corresponding closing bracket, but not before more deeply nested pairs have been closed, the RNNs do not have to explicitly learn that the principle holds true at any nesting depth and at any character distance. Conversely, training RNNs mostly on complex words leads to an *overspecification* of the learned rules: the model rarely encounters the underlying principle of D_2 in a simple form and might assume that it only holds true for specific nesting depths or distances, leading to a needlessly - and inaccurately - complex rule set. This interpretation is similar to what Zeng et al. (1993) have found when experimenting with incrementally increasing the length of strings in training: Analog RNNs have been shown to learn 'soft' solutions that are then incrementally hardened as more restrictions are necessary.

6 Conclusion

This work has set out to answer three questions, as posed in Chapter 1. Related literature has been consulted to choose a proper approach. However, current literature contains neither a benchmark dataset to train and test models on, nor a unified set of tasks and measures to do so. Due to these facts, most results in current literature discussing model performance on D_2 are incomparable to each other.

To assess model performance, I have adapted the two experiments proposed by Bernardy (2018) for a classification task. They were explicitly designed to investigate model performance on long-range dependencies, as well as a model's ability to generalize to deeper nesting depths. I have reported accuracy, precision, recall and the compound measure of F1 score across models and experiments, as well as providing a closer look at common error sources for the models.

In addition to assessing three different architectures on two experiments, I have investigated the impact of hidden unit number and training corpus composition on model performance. For the former, each architecture was implemented in 9 different models with hidden unit number $n \in \{2^1, 2^2, \ldots, 2^9\}$. All models have been trained with the same hyperparameters. To achieve the latter, I have constructed three training corpora: a baseline corpus (Base), a corpus containing words with a high nesting depth and maximum bracket distance (High LRD) and a corpus containing words with a low nesting depth and maximum bracket distance (Low LRD). All corpora contained 1,000,000 words, 500,000 of which were correct. The incorrect words consisted of 250,000 words with extra open and 250,000 words with extra closing brackets. In total, 81 models were trained, and each model was evaluated on both experiment data sets.

The results of both experiments show learning D_2 to be a task of not trivial and highly volatile difficulty: the number of hidden units in models with an accuracy well above random guessing ranges from 2 to 512. A vast majority of models failed to extract any useful information from the training data, either staying near the random decision baseline or vastly underperforming. Successful models mostly exhibit a behaviour indicating they have extracted a basic approximation of D_2 from the linear training data by way of barely differentiating between the two error categories.

In total, processing extreme long-range dependencies spanning the whole word was more difficult for the models than processing extreme unseen nesting depths. This might be owed to the fact that most models did not learn a valid representation of D_2: the unseen nesting depths still featured fairly short long-range dependencies compared to Experiment 1, which can easily be resolved in memory without needing to understand recursion.

As expected, LSTMs and GRUs outperformed SRNNs. While successful LSTM models achieved the highest individual accuracy scores (topping out at 99.3%) and came closest to a perfect open/closed misclassification ratio of 1, GRUs were more consistently successful, both producing the most models performing better than chance and achieving the highest average accuracy. It stands to reason that, while not succeeding under the circumstances of this study, LSTMs show the highest capability of perfectly learning D_2. Taking the Chomsky-Schützenberger Representation Theorem (Chomsky and Schützenberger (1963)) into account, it seems possible that these architectures are capable of learning the vast class of Type-2 languages - which likely contains natural language.

I have found corpus complexity to have a significant impact on model performance. Models trained on the Low LRD corpus largely outperformed the Base corpus models, while High LRD corpus models underperformed. While models trained on complex words failing to generalize to longer, more complex words seems paradoxical at first, these results suggest that whatever rules RNNs extract from the input data, rule extraction becomes harder the more complex the data is. Taking into account the open/closed bracket misclassification ratio, I suggested that RNNs can extract a small number of simple rules from simple data and generally apply them in a more complex context. The more complex training data encourages the models to instead learn a larger set of overly specified rules - and then fail to generalize them. If a Low LRD trained model learns that an open bracket must always be closed by its corresponding closing bracket, the High LRD trained model might learn that an open bracket may be closed by its corresponding closing bracket only after a certain number of characters, or only after a certain number of nesting levels have been resolved, leading to a bloated, overspecified and misleading ruleset.

6.1 Further Research

The most compelling result of this research is the effect of training corpus complexity on generalizability. While it holds true in this case, formal language data is by definition rigorously structured and, for D_2, rather simple and limited. Natural language data features a far bigger alphabet, complex syntactic, semantic and morphological dependencies and irregularities. Nonetheless, RNNs for NLP tasks improving with structurally simplified training data poses an intriguing and possibly fruitful avenue of future research. Furthermore, while LSTMs emerged as the most promising architecture to learn D_2, they have proven to be more volatile than GRUs. Whether there is an inherent difference between the architectures remains unclear and may be explored with more established and sophisticated methods of internal state analysis.

Bibliography

Autebert, J.-M., Berstel, J., and Boasson, L. (1997). Context-Free Languages and Pushdown Automata. In *Handbook of Formal Languages: Volume 1 Word, Language, Grammar*, pages 111–174. Springer Berlin Heidelberg, Berlin, Heidelberg.

Bach, E., Brown, C., and Marslen-Wilson, W. (1986). Crossed and nested dependencies in German and Dutch: A psycholinguistic study. *Language and Cognitive Processes*, 1:249–262.

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.

Bernardy, J.-P. (2018). Can recurrent neural networks learn nested recursion? *LiLT (Linguistic Issues in Language Technology)*, 16(1).

Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar. Association for Computational Linguistics.

Chomsky, N. (1959). On certain formal properties of grammars. *Information and Control*, 2(2):137 – 167.

Chomsky, N. and Schützenberger, M. (1963). The Algebraic Theory of Context-Free Languages*. In *Computer Programming and Formal Systems*, volume 35 of *Studies in Logic and the Foundations of Mathematics*, pages 118 – 161. Elsevier.

Cleeremans, A., Servan-Schreiber, D., and Mcclelland, J. (1989). Finite State Automata and Simple Recurrent Networks. *Neural Computation - NECO*, 1:372–381.

Deleu, T. and Dureau, J. (2016). Learning Operations on a Stack with Neural Turing Machines. *CoRR*, abs/1612.00827.

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

Fitch, W. T., Friederici, A. D., and Hagoort, P. (2012). Pattern perception and computational complexity: introduction to the special issue. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 367(1598):1925–1932.

Frank, S. L., Trompenaars, T., and Vasishth, S. (2016). Cross-Linguistic Differences in Processing Double-Embedded Relative Clauses: Working-Memory Constraints or Language Statistics? *Cognitive Science*, 40(3):554–578.

Gers, F. A. and Schmidhuber, E. (2001). LSTM recurrent networks learn simple context-free and context-sensitive languages. *IEEE Transactions on Neural Networks*, 12(6):1333–1340.

Hagège, C. (1976). Relative Clause, Center-Embedding, and Comprehensibility. *Linguistic Inquiry*, 7(1):198–201.

Hamilton, H. W. and Deese, J. (1971). Comprehensibility and subject-verb relations in complex sentences. *Journal of Verbal Learning and Verbal Behavior*, 10(2):163 – 170.

Hochreiter, S. (1998). The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6:107–116.

Hochreiter, S. and Schmidhuber, J. (1997). Long Short-term Memory. *Neural computation*, 9:1735–80.

Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2006). *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

Joulin, A. and Mikolov, T. (2015). Inferring Algorithmic Patterns with Stack-Augmented Recurrent Nets. *CoRR*, abs/1503.01007.

Jurafsky, D. and Martin, J. H. (2009). *Speech and Language Processing (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Karlsson, F. (2007). Constraints on Multiple Center-Embedding of Clauses. *Journal of Linguistics*, 43(2):365–392.

Kingma, D. and Ba, J. (2014). Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations*.

Koutník, J., Greff, K., Gomez, F., and Schmidhuber, J. (2014). A clockwork RNN. *31st International Conference on Machine Learning, ICML 2014*, 5.

Li, T., Rabusseau, G., and Precup, D. (2018). Nonlinear Weighted Finite Automata. In Storkey, A. and Perez-Cruz, F., editors, *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*, volume 84 of *Proceedings of Machine Learning Research*, pages 679–688, Playa Blanca, Lanzarote, Canary Islands. PMLR.

McKinney, W. (2011). pandas: a foundational Python library for data analysis and statistics. *Python for High Performance and Scientific Computing*, 14.

Newmeyer, F. and Preston, L. (2014). *Measuring Grammatical Complexity*. Oxford linguistics. Oxford University Press.

Oliphant, T. E. (2006). *A guide to NumPy*, volume 1. Trelgol Publishing USA.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Petersson, K. M. and Hagoort, P. (2012). The neurobiology of syntax: Beyond string sets. *Philosophical transactions of the Royal Society of London. Series B, Biological sciences*, 367:1971–83.

Rodriguez, P. and Wiles, J. (1998). Recurrent Neural Networks Can Learn to Implement Symbol-sensitive Counting. In *Proceedings of the 1997 Conference on Advances in Neural Information Processing Systems 10*, NIPS '97, pages 87–93, Cambridge, MA, USA. MIT Press.

Sennhauser, L. and Berwick, R. C. (2018). Evaluating the ability of LSTMs to learn context-free grammars. *CoRR*, abs/1811.02611.

Shieber, S. M. (1987). Evidence Against the Context-Freeness of Natural Language. In *The Formal Complexity of Natural Language*, pages 320–334. Springer Netherlands, Dordrecht.

Sipser, M. (2013). *Introduction to the Theory of Computation*. Course Technology, Boston, MA, third edition.

Skachkova, N., Trost, T., and Klakow, D. (2018). Closing Brackets with Recurrent Neural Networks. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 232–239, Brussels, Belgium. Association for Computational Linguistics.

Suzgun, M., Gehrmann, S., Belinkov, Y., and Shieber, S. M. (2019). LSTM Networks Can Perform Dynamic Counting. *CoRR*, abs/1906.03648.

Williams, R. J. and Zipser, D. (1995). Gradient-based learning algorithms for recurrent networks and their computational complexity. In *Backpropagation: Theory, Architectures, and Applications*, page 433–486. L. Erlbaum Associates Inc., USA.

Yu, X., Vu, N. T., and Kuhn, J. (2019). Learning the Dyck Language with Attention-based Seq2Seq Models. In *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 138–146, Florence, Italy. Association for Computational Linguistics.

Zeng, Z., Goodman, R. M., and Smyth, P. (1993). Learning Finite State Machines With Self-Clustering Recurrent Networks. *Neural Computation*, 5:976–990.

Zeng, Z., Goodman, R. M., and Smyth, P. (1994). Discrete recurrent neural networks for grammatical inference. *IEEE transactions on neural networks*, 5 2:320–30.

s

Appendix

```
# generate_raw_data.py
# Script to sample words from Dyck(2) according to Sennhauser &
    Berwick 2018.
#
# Grammar:
# S -> Z S | Z
# Z -> B | T
# B -> [ S ] | { S }
# T -> [] | {}
#
# branch = Z -> B
# concat = S -> Z S
# s(l) = min(1, -3 * (l/n) + 3)
# P_branch = r_b * s(l); r_b ~ U(0.4, 0.8)
# P_concat = r_c * s(l); r_c ~ U(0.4, 0.8)
# l = number of already generated characters in the sentence, n =
    total goal length
# https://en.wikipedia.org/wiki/Uniform_distribution_(continuous) == U
    (0.4, 0.8)

import sys
import numpy as np
import os

LIMIT = int(sys.argv[1])
GOAL_CORPUS_SIZE = int(sys.argv[2])

def countCharacters(word):
  '''Counts the number of generated terminal symbols in a string.
      args:
        word: string, string of terminal and non-terminal symbols.
      returns:
        generated_characters: int, number of generated terminal
    symbols in word.'''
  generated_characters = 0
  for character in word:
    if character in {"[","]","{","}"}:
      generated_characters +=  1

  # The beginning 'S' is counted as 2 generated terminal symbols here
    - if the word contains an 'S', the minimum amount of terminal
    symbols in the fully derived word is 2
  return generated_characters + 2
```

```

def findNonterminals(word):
  '''Assesses whether the input string contains any non-terminal symbols.
      args:
        word: string, string of terminal and non-terminal symbols.
      returns:
        bool'''
  if "S" in word:
    return True
  elif "Z" in word:
    return True
  elif "T" in word:
    return True
  elif "V" in word:
    return True
  elif "B" in word:
    return True
  else:
    return False

def expand(word, limit):
  '''Expands the input word by applying probabilistic grammar rules. The generated word cannot be longer than the given limit.
      args:
        word: string, string of terminal and non-terminal symbols.
        limit: int, maximum length the word is allowed to have.
      returns:
        word: string, string of terminal and non-terminal symbols.'''
  # Sampling probabilities as per Sennhauser & Berwick 2018, once per word
  prob_b = np.random.uniform(0.6, 0.9)
  prob_c = np.random.uniform(0.6, 0.9)
  for i in range(len(word)):
    # Once word length has exceeded the limit, the candidate is invalid and will not be processed further
    if len(word) > limit:
      return word
    else:

      s_l = min(1, -3 * countCharacters(word)/float(limit) + 3)
      P_branch = prob_b * s_l
      P_concat = prob_c * s_l
      random = np.random.uniform(0.0, 1.0)

      # Applying probabilistic Dyck2 grammar by replacing left-hand non-terminals with their right-hand production
      if word[i] == 'S':
```

```
        if P_concat <= 0.0:
          word = word[:i] + 'Z' + word[i+1:]
        elif random < P_concat:
          word = word[:i] + 'ZS' + word[i+1:]
        else:
          word = word[:i] + 'Z' + word[i+1:]
      elif word[i] == 'Z':
        if P_branch <= 0.0:
          word = word[:i] + 'T' + word[i+1:]
        elif random < P_branch:
          word = word[:i] + 'B' + word[i+1:]
        else:
          word = word[:i] + 'T' + word[i+1:]
      elif word[i] == 'B':
        if random < 0.5:
          word = word[:i] + '[S]' + word[i+1:]
        else:
          word = word[:i] + '{S}' + word[i+1:]
      elif word[i] == 'T':
        if random < 0.5:
          word = word[:i] + '[]' + word[i+1:]
        else:
          word = word[:i] + '{}' + word[i+1:]
      else:
        continue

  return word

def createCorpus(limit, goal_corpus_size):
  '''Generates a corpus of size goal_corpus_size of words with len <=
   limit. Saves the result in steps of 10000 generated words as a txt
    file. Saves a final corpus as a txt file.
      args:
        limit: int, maximum length for a word in the corpus.
        goal_corpus_size: int, number of words in the full corpus.
      returns:
        none'''
  counter = 0
  corpus = set()
  # Populates the corpus
  while len(corpus) < goal_corpus_size:
    prev_save_len = 0
    counter += 1
    word = 'S'
    # Expands the word until it either is fully derived or exceeds the
    desired length
    while findNonterminals(word) and len(word) <= limit:
      word = expand(word, limit)
```

```
        # If the word is fully derived and fulfills the length requirement, it is added to the corpus
        if not findNonterminals(word):
            if len(word) <= limit:
                old_length = len(corpus)
                corpus.add(word)

        # Print occasional update on the generation process to the console
        if counter % 50000 == 0:
            print("Corp Size: {}\tGen words: {}".format(len(corpus), counter))

        # Save interim results
        if len(corpus) % 10000 == 0 and len(corpus) != prev_save_len:
            print("Saving corpus, length {} ...".format(len(corpus)))
            filename = os.path.join('..', 'corpus', "cumlen{}_{}.txt".format(limit, len(corpus)))
            outfile = open(filename, 'w')
            for item in corpus:
                outfile.write(item)
                outfile.write('$')
            outfile.close()

    # Save full corpus
    filename = os.path.join('..', 'corpus', "cumlen{}_{}.txt".format(limit, len(corpus)))
    outfile = open(filename, 'w')
    for item in corpus:
        outfile.write(item)
        outfile.write('$')
    outfile.close()

createCorpus(LIMIT, GOAL_CORPUS_SIZE)
```

Listing 1: Script to generate the base data from which the corpora are built. Sampling is done according to Sennhauser and Berwick (2018).

```python
# corpus_tools.py
# Works with a base file of correct Dyck words to create
# Generates classification datasets with a 50/50 split on correct/incorrect words each from a basic sampling of correct Dyck(2) words.
# Creates following datasets:
# TRAINING
#   - base
# - high LRD
# - low LRD
# EXPERIMENTS
# - LRD
# - ND

import os
import sys
import random
import pickle
import numpy as np

INPUT_FILE = 'cumlen20_1180000.txt'
INPUT_PATH = os.path.join('..', 'corpus', INPUT_FILE)
SIZE = 1000000 # Training Corpus Size Target
LENGTH = int(INPUT_FILE[6:8]) # max length of word in set
POSITIVE_RATIO = 0.5
NEGATIVE_RATIO = 1-POSITIVE_RATIO
BD_CUTOFF = 19
MAX_BD_CUTOFF = 10
OUTPUT_TRAINING = os.path.join('..', 'training', 'base.csv')
OUTPUT_HIGH_LRD = os.path.join('..', 'training', 'high.csv')
OUTPUT_LOW_LRD = os.path.join('..', 'training', 'low.csv')
OUTPUT_LRD = os.path.join('..', 'experiment', 'LRD.csv')
OUTPUT_ND = os.path.join('..', 'experiment', 'ND.csv')

def samePair(char1, char2):
  '''Helper function to check whether the two characters are a valid bracket pair.
    args:
      char1: string, character in word
      char2: string, character in word
    returns:
      bool'''
  if char1 == '{' and char2 == '}':
    return True
  elif char1 == '[' and char2 == ']':
    return True
  else:
```

```
    return False

def maxNestingDepth(word):
  '''Calculates the maximum nesting depth within a D_2 word.
      args:
        word: string, Dyck word consisting of [, {, }, ] as brackets.
      returns:
        max_depth: int'''
  # Remove EOW symbol when working with processed corpus
  if word[-1] == '$':
    word = word[:-1]

  max_depth = 0
  depth = 0
  for character in word:
    if character == "[" or character == "{":
      depth += 1
    else: # Any other character must be a closing bracket and thus reduce depth
      depth -= 1
    if depth < 0:
      return -1 # Indicates a corrupted word with a superfluous closing bracket in analysis

    if depth > max_depth:
      max_depth = depth

  return max_depth

def maxValidNestingDepth(word):
  '''Calculates the maximum valid nesting depth within a word -- if the word was corrupted, negative nesting depth might occur.
  This function disregards that.
      args:
        word: string, Dyck word consisting of [, {, }, ] as brackets.
      returns:
        max_depth: int'''
  # Remove EOW symbol when working with processed corpus
  if word[-1] == '$':
    word = word[:-1]

  max_depth = 0
  depth = 0
  for character in word:
    if character == "[" or character == "{":
      depth += 1
    elif character == "]" or character == "}":
      depth -= 1
```

```
    else:
      continue
    if depth < 0:
      continue

    if depth > max_depth:
      max_depth = depth

  return max_depth

def nestingDepthAtPosition(word, position):
  '''Calculates the nesting depth of the character at word[position].
      args:
        word: string, Dyck word consisting of [, {, }, ] as brackets.
        position: int, index for word.
      returns:
        depth: int'''
  depth = 0
  for i in range(position):
    if word[i] == "[" or word[i] == "{":
      depth += 1
    else:
      depth -= 1

  return depth

def maxBracketDistance(word):
  '''returns maximum distance in characters between an opening and its corresponding closing bracket, i.e. maxBracketDistance('[{[]}]') is 4.
    args:
      word: string, Dyck word consisting of [, {, }, ] as brackets.
    returns:
      max(max_distance_square,max_distance_curly): int, maximum distance for either of the two bracket pair types = maximum distance in the word.'''
  # Remove EOW symbol when working with processed corpus.
  if word[-1] == '$':
    word = word[:-1]
  distance_square = 0
  max_distance_square = 0
  distance_curly = 0
  max_distance_curly = 0
  square_stack = []
  curly_stack = []

  # Counts 'seen' characters between two corresponding brackets
  for character in word:
```

```
    # Fill/empty the two stacks to keep count of unclosed brackets
    if character == "[":
      square_stack.append(character)
    elif character == "{":
      curly_stack.append(character)
    elif character == "]":
      square_stack.pop()
    else:
      curly_stack.pop()

     # Check if stack is empty - means all opening brackets have been closed
    if not square_stack:
      # Save new max_distance_square and reset 'seen' characters counter
      if distance_square > max_distance_square:
        max_distance_square = distance_square - 1 # Adjust for first character being counted
      distance_square = 0
    else: # Increment 'seen' characters counter as long as there is still an unclosed bracket on the stack
      distance_square += 1

    # Same process as for square_stack
    if not curly_stack:
      if distance_curly > max_distance_curly:
        max_distance_curly = distance_curly - 1
      distance_curly = 0
    else:
      distance_curly += 1

  return max(max_distance_square, max_distance_curly)

def bracketDistanceAtPosition(word, position):
  '''returns distance from closing bracket at a given position to
  its corresponding opening bracket.
    args:
      word: string, Dyck word consisting of [, {, }, ] as brackets.
      position: position: int, index for word.
    returns:
      distance: int, number of characters between word[position] and the corresponding opening bracket.'''
  distance = 0
  stack = []
  character = word[position]

  # Make sure a closing bracket is at the position. Otherwise, it's an opening bracket, for which the measure does not make sense
```

```
  if character == "]":
    opener = "["
  elif character == "}":
    opener = "{"
  else:
    return 0

  # Go backwards through the word, starting at position. Same logic applies as in maxBracketDistance(word)
  for i in range(position, -1, -1):
    if word[i] == opener:
      stack.pop()
    elif word[i] == character:
      stack.append(character)

    if not stack:
      return distance-1
    else:
      distance += 1

def determineError(word):
  '''Given a corrupted D_2 word, this function determines the kind of error - too many opening or too many closing brackets.
    args:
      word: corrupted Dyck-2 word
    returns:
      error: type of error'''
  open = ('[', '{')
  closed = (']', '}')
  o = 0
  c = 0
  for character in word:
    if character in open:
      o+=1
    elif character in closed:
      c+=1
    else:
      continue
  error = o-c
  if error < 0:
    return 'closed'
  elif error > 0:
    return 'open'
  else:
    return 'none'

def findErrorPosition(word):
  '''Given a corrupted D_2 word, this function determines the position
```

```
    of the corrupted bracket.
    args:
      word: corrupted Dyck-2 word
    returns:
      position: character position of corrupted bracket'''
  word = word.lstrip('0')
  # Stacks are filled with character positions. If a stack is not empty by the time the word is over,
  # the remaining position is the error position.
  # If there is an attempt to pop an empty stack before the word is over, the current position is the error position.
  square_stack = []
  curly_stack = []

  for i in range(len(word)):
    if word[i] == '[':
      square_stack.append(i)
    elif word[i] == '{':
      curly_stack.append(i)
    elif word[i] == ']':
      if square_stack:
        square_stack.pop()
      else:
        return i
    elif word[i] == '}':
      if curly_stack:
        curly_stack.pop()
      else:
        return i
    else:
      continue
  if square_stack:
    return square_stack[0]
  elif curly_stack:
    return curly_stack[0]
  else:
    raise ValueError("Encountered unknown corruption in word\n{}".format(word))

def measureLength(corpus):
  '''Calculates average length and variance thereof for all words in the corpus.
    args:
      corpus: list of strings, list of Dyck words.
    returns:
      avg: float, average length of words in corpus.
      var: float, variance of length of words in corpus.'''
  if len(corpus) == 0:
```

```
    return 0., 0.
  total = []
  for word in corpus:
    total.append(len(word))
  avg = sum(total)/float(len(total))
  var = sum((length - avg)**2 for length in total)/float(len(total))

  return avg, var

def measureMaxNestingDepth(corpus):
  '''Calculates average maximum nesting depth and variance thereof for all words in the corpus.
    args:
      corpus: list of strings, list of Dyck words.
    returns:
      avg: float, average maximum nesting depth of words in corpus.
      var: float, variance of maximum nesting depth of words in corpus.'''
  if len(corpus) == 0:
    return 0., 0.
  total = []
  for word in corpus:
    total.append(maxNestingDepth(word))
  avg = sum(total)/float(len(total))
  var = sum((depth - avg)**2 for depth in total)/float(len(total))

  return avg, var

def measureMaxBracketDistance(corpus):
  '''Calculates average maximum bracket distance and variance thereof for all words in the corpus.
    args:
      corpus: list of strings, list of Dyck words.
    returns:
      avg: float, average maximum bracket distance of words in corpus.
      var: float, variance of maximum bracket distance of words in corpus.'''
  if len(corpus) == 0:
    return 0., 0.
  total = []
  for word in corpus:
    total.append(maxBracketDistance(word))
  avg = sum(total)/float(len(total))
  var = sum((dist - avg)**2 for dist in total)/float(len(total))

  return avg, var

def evaluateCorpus(corpus):
```

```
    '''Calculates average and variance for three measures over all words in a corpus: word length, maximum nesting depth and maximum bracket distance.
    args:
      corpus:
    returns:
      3 x (avg, var): 3 tuples of floats, average and variance for the respective measure.'''
    corpus = [entry[0][:-1] for entry in corpus if entry[1]] # Removing EOW

    return measureLength(corpus), measureMaxNestingDepth(corpus), measureMaxBracketDistance(corpus)

def printStats(size, avgLen, varLen, avgMaxND, varMaxND, avgMaxBD, varMaxBD):
    '''Prints a table with all calculated corpus stats to the console. Table is for copy-pasting into .tex file.
    args:
      size: int, number of words in the corpus
      avgLen: float, average length of word in the corpus
      varLen: float, variation of length of words in the corpus
      avgMaxND: float, average maximum nesting depth of word in the corpus
      varMaxND: float, variation of maximum nesting depth of words in the corpus
      avgMaxBD: float, average maximum bracket depth of word in the corpus
      varMaxBD: float, variation of maximum bracket depth of words in the corpus
    returns:
      none'''
    print("Size\tAvg Length \t Avg MaxNestDepth \t Avg MaxBrackDist")
    print("{}\t${:3.2f}$ (${:3.2f}$) & ${:3.2f}$ (${:3.2f}$) & ${:3.2f}$ (${:3.2f}$)".format(size, avgLen, varLen, avgMaxND, varMaxND, avgMaxBD, varMaxBD))

def largerBD(corpus):
    '''Increases average bracket distance for a corpus by finding words with a low maximum bracket distance, deleting the lowest distance pair from the word and then wrapping the word in a matching pair of brackets.
    args:
      corpus: list of strings, list of Dyck words.
    returns:
      corpus: list of strings, list of Dyck words.'''
    # 'Word collectors' are initialized as lists to allow iteration through them
```

```
  small_BD = []
  big_BD = []

  # Find words with low maximum bracket distance
  for entry in corpus:
    word = entry[0][:-1]
    correct = entry[1]
    if correct:
      maxBD = maxBracketDistance(word)
      if maxBD < BD_CUTOFF:
        small_BD.append(word)
      elif maxBD >= BD_CUTOFF:
        big_BD.append(word)

  # Modify low maximum bracket distance words
  for word in small_BD:
    prev_ND = 0 # Nesting depth to compare to
    for i in range(len(word)):
      # Calculate nesting depth at each position of the word. Once it decreases, a closing bracket has been found
      ND = nestingDepthAtPosition(word, i)
      if ND < prev_ND:
        # Check if this position belongs to a bracket pair eligible for deletion - only {} and [] are eligible, since they have the shortest possible bracket distance
        char = word[i-1]
        prev_char = word[i-2]
        if samePair(prev_char, char):
          bracket = random.randint(0, 1)
          if bracket:
            word = '[' + word[:i-2] + word[i:] + ']'
            break
          else:
            word = '{' + word[:i-2] + word[i:] + '}'
            break
      prev_ND = ND # Update nesting depth
    big_BD.append(word) # Populate modified list

  BD_set = set(big_BD) # Fast deletion of duplicates.
  BD_list = list(BD_set)
  incorrect = [entry for entry in corpus if entry[1]==0]
  BD = [[word+'$',1] for word in BD_list] # Complete newly created and old correct words with EOW and class
  BD = BD + incorrect
  random.shuffle(BD)

  return BD

```

```python
def smallerBD(corpus):
  '''Decreases average bracket distance for a corpus by finding words with a high maximum bracket distance. In those words, the pair with the highest maximum bracket distance is found. The opening bracket is then moved right in front of the closing bracket.
   args:
     corpus: list of strings, list of Dyck words.
   returns:
     corpus: list of strings, list of Dyck words.'''
  # 'Word collectors' are initialized as lists to allow iteration through them
  big_BD = []
  small_BD = []

  # Find words with high maximum bracket distance
  for entry in corpus:
    word = entry[0][:-1]
    correct = entry[1]
    if correct:
      maxBD = maxBracketDistance(word)
      if maxBD > MAX_BD_CUTOFF:
        big_BD.append(word)
      elif maxBD <= MAX_BD_CUTOFF:
        small_BD.append(word) # big_BD only features word entries, since all big_BD entries are correct

  for word in big_BD:
    max_BD = 0
    max_pos = 0
    for i in range(len(word)):
      # Calculate bracket distance at each position of the word. Once the maximum bracket distance has been found, the position of the closing bracket is recorded
      BD = bracketDistanceAtPosition(word, i)
      if BD > max_BD:
        max_pos = i
        max_BD = BD
    closer = max_pos # Position of longest distance closing bracket
    opener = max_pos - max_BD - 1 # Fix off by one return of bd@pos
    # New word is created by deleting the opener from its original position and moving it right in front of the closing bracket
    # This ensures grammaticality and reduces maxBracketDistance
    new_word = word[:opener] + word[opener+1:closer] + word[opener] + word[closer:]
    small_BD.append(new_word)

  BD_list = list(set(small_BD))
  incorrect = [entry for entry in corpus if entry[1]==0]
```

```
  BD = [[word+'$',1] for word in BD_list] # Complete newly created and old correct words with EOW and class
  BD = BD + incorrect

  return BD

def corrupt_words(correct_corpus, mode):
  '''Turns correct Dyck_2 words into incorrect ones by replacing a random opening bracket with a random closing one or vice versa, depending on the mode.
    args:
      correct_corpus: list of correct word-class tuples
      mode: string. "open"/"close" - determines brackets being changed
    returns:
      incorrect: list of incorrect word-class tuples'''
  incorrect = set() # set ensures uniqueness
  if mode == "open":
    replace = ("{","[")
    replacement = ["]","}"]
  else:
    replace = ("}","]")
    replacement = ["{","["]

  for entry in correct_corpus:
    word = entry[0]
    changed = 0
    c = 0
    while not changed:
      c += 1
      id = random.randint(0, len(word)-1)
      if word[id] in replace:
        closing = random.choice(replacement)
        new_word = word[:id] + closing + word[id+1:]
        assert len(new_word)==len(word)
        incorrect.add(new_word)
        changed = 1
      elif c >= 1000000:
        break
  incorrect = [[word,0] for word in incorrect]

  return incorrect

def create_LRD(base, subword_length):
  '''Creates the dataset for the extreme long range dependency (LRD) experiment/experiment 1.
    args:
      base: list of correct word-class tuples
    returns:
```

```
      LRD: list of correct LRD word-class tuples'''
  LRD = set()
  base = [entry[0] for entry in base if len(entry[0])==subword_length]
  for i in range(len(base)*5):
    bracket = random.randint(0,1)
    w1 = random.sample(base, 1)[0][:-1]
    w2 = random.sample(base, 1)[0][:-1]
    if bracket:
      new_word = '[' + w1 + w2+ ']$'
    else:
      new_word = '{' + w1 + w2 + '}$'
    LRD.add(new_word)
  LRD = [[word,1] for word in LRD]

  return LRD

def create_ND(base, infix_length):
  '''Creates the dataset for the extreme new depths (ND) experiment/
   experiment 2.
    args:
      base: list of correct word-class tuples
    returns:
      ND: list of correct LRD word-class tuples'''
  ND = set()
  base = [entry[0] for entry in base if len(entry[0])==infix_length]
  for i in range(len(base)*5):
    new_word = random.sample(base, 1)[0][:-1]
    for i in range(5):
      bracket = random.randint(0,1)
      if bracket:
        new_word = '[' + new_word + ']'
      else:
        new_word = '{' + new_word + '}'
    ND.add(new_word + '$')
  ND = [[word,1] for word in ND]

  return ND

def create_corpus(data, type):
  '''Creates full corpora for a classification task. Increases or
   lowers bracket distance for the high/low LRD training corpora.
    Prints corpus stats to the console, with the values in LaTeX
   formatting.
      args:
        data: list of [word,correct-bool] pairs, filled with correct
   generated D_2 words
        type: string, high/low/[misc], triggers increasing/decreasing/
   disregarding average bracket distance across the corpus
```

```
      returns:
        corpus: list of [word,correct-bool] pairs, filled with a
    1:0.5:0.5 ratio of correct:incorrect_open:incorrect_closed words
    '''
  if type == 'high':
    corpus_correct = largerBD(data)
  elif type == 'low':
    corpus_correct = smallerBD(data)
  else:
    corpus_correct = data
  corpus_incorrect_open = corrupt_words(corpus_correct, 'open')
  corpus_incorrect_closed = corrupt_words(corpus_correct, 'closed')
  # Debug print
  #print(" === {} ===\nCorrect\tIncorrect O\tIncorrect C\n{}\t{}\t{}".
    format(type.upper(), len(data), len(corpus_incorrect_open), len(
    corpus_incorrect_closed)))
  corpus = data[:int(POSITIVE_RATIO*SIZE)] + corpus_incorrect_open[:
    int(NEGATIVE_RATIO/2.*SIZE)] + corpus_incorrect_closed[:int(
    NEGATIVE_RATIO/2.*SIZE)]
  random.shuffle(corpus)
  (avLen, varLen), (avMaxND, varMaxND), (avMaxBD, varMaxBD) =
    evaluateCorpus(corpus)
  printStats(len(corpus), avLen, varLen, avMaxND, varMaxND, avMaxBD,
    varMaxBD)

  return corpus


def save2file(outpath, corpus):
  '''Saves the corpus as a .csv file to the specified path.
      args:
        outpath: filepath of the output file
        corpus: list of [word,correct-bool] pairs, to be used for
    training/experiments on RNNs
      returns:
        none'''
  outfile = open(outpath, 'w')
  outfile.write('word,value\n')
  for entry in corpus:
    outfile.write('{},{}\n'.format(entry[0],entry[1]))

def create_corpora():
  '''Creates all datasets needed for the classification tasks from the
     input file.
      args:
        none
      returns:
        none'''
```

```
file = open(INPUT_PATH, 'r')
EOW = '$'
raw_text = file.read()

print("Creating Base...")
raw_classified = [[word+EOW,1] for word in raw_text.split(EOW)]

print("Corrupting Base...")
base = create_corpus(raw_classified, 'base')
save2file(OUTPUT_TRAINING, base)

print("Corrupting High LRD...")
highLRD = create_corpus(raw_classified, 'high')
save2file(OUTPUT_HIGH_LRD, highLRD)

print("Corrupting Low LRD...")
lowLRD = create_corpus(raw_classified, 'low')
save2file(OUTPUT_LOW_LRD, lowLRD)

print("Creating LRD...")
LRD_correct = create_LRD(raw_classified, LENGTH-2+1)
LRD = create_corpus(LRD_correct, 'LRD')
save2file(OUTPUT_LRD, LRD)

print("Creating ND...")
ND_correct = create_ND(raw_classified, LENGTH+1)
ND = create_corpus(ND_correct, 'ND')
save2file(OUTPUT_ND, ND)
```

Listing 2: Collection of functions to measure corpus and word properties. Used to generate five classification datasets: three different training sets and the two experiment sets.

```
# RNN_classifier.py
# Training, testing and experimenting on RNN models

import os
import sys
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
import tensorflow as tf
tf.keras.backend.clear_session() # Reduces memory leak during training - seems to be issue with TF2.0
from tensorflow.keras import layers
from tensorflow.keras import losses
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Activation, Dense, Embedding, SimpleRNN, GRU
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing import sequence
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint

MODE = sys.argv[1]
NETWORK = sys.argv[2]
HIDDEN_UNITS = int(sys.argv[3])
CORPUS = sys.argv[4]
EXPERIMENT = sys.argv[5]
EPOCHS = 100
TEST_SIZE = 0.2
VOCAB_SIZE = 7 # 6 characters + 1 do combat off by 1 error in Embedding Layer
MAX_LEN = ((20-2)*2)+1+2 # Maximum length of words in all corpora and experiments -- LRD length formula
BATCH = 512
CHECKPOINT_DIR = os.path.join('..', 'saved_models', CORPUS, NETWORK, str(HIDDEN_UNITS))
CORPUS_DF = pd.read_csv('../training/{}.csv'.format(CORPUS),delimiter=',',encoding='latin-1')
EXP_DF = pd.read_csv('../experiment/{}.csv'.format(EXPERIMENT))

# ============ PROCESSING INPUT DATA ============

def filter_length(df, max_len_train):
  '''Filters words of length > max_len_train out of a dataframe.
    args:
      df: pd.DataFrame of words
```

```
      max_len_train: Maximum length words in df should have
    returns:
      df: pd.DataFrame of words'''
  mask = (df['word'].str.len() <= max_len_train)
  df = df.loc[mask]
  df.head()
  df.info()

  return df

def prepare_training_data(df, max_word_length=MAX_LEN):
  '''Preprocesses training data by splitting it into train/test and
   transforming the strings into sequences of numbers.
    args:
      df: pd.DataFrame of words
      max_word_length: Maximum wanted length of words
    returns:
      X_train: Preprocessed words for the train split
      X_test: Preprocessed words for the test split
      Y_train: Target values for the train split
      Y_test: Target values for the test split
      tok: Tokenizer fitted on training corpus, to be used on
   experiment corpus'''
  df = filter_length(df, max_word_length)
  X = df.word
  Y = df.value
  # Encode target values
  le = LabelEncoder()
  Y = le.fit_transform(Y)
  Y = Y.reshape(-1,1)

  X_train_temp,X_test_temp,Y_train,Y_test = train_test_split(X,Y,
   test_size=TEST_SIZE)

  # Preprocess words into sequences of numbers corresponding to the
   characters
  tok = Tokenizer(char_level=True)
  tok.fit_on_texts(X_train_temp)
  sequences = tok.texts_to_sequences(X_train_temp)
  X_train = sequence.pad_sequences(sequences,maxlen=MAX_LEN)
  test_sequences = tok.texts_to_sequences(X_test_temp)
  X_test = sequence.pad_sequences(test_sequences,maxlen=MAX_LEN)

  return X_train, X_test, Y_train, Y_test, tok

def prepare_experiment_data(dataframe, tok, max_word_length=MAX_LEN):
  '''Preprocesses experiment data by splitting it into train/test and
   transforming the strings into sequences of numbers.
```

```
    args:
      df: pd.DataFrame of words
      max_word_length: Maximum wanted length of words
      tok: Tokenizer as fitted on the training corpus
    returns:
      X: Preprocessed words
      Y: Target values'''
  df = filter_length(dataframe, max_word_length)
  X_temp = df.word
  Y = df.value
  # Encode target values
  le = LabelEncoder()
  Y = le.fit_transform(Y)
  Y = Y.reshape(-1,1)

  # Preprocess words into sequences of numbers corresponding to the characters
  # The same tokenizer is used for training and experiment data (training tokenizer is called in this function)
  sequences = tok.texts_to_sequences(X_temp)
  X = sequence.pad_sequences(sequences,maxlen=MAX_LEN)

  return X, Y

# ============ MODELS ============

def build_model(layer_size=HIDDEN_UNITS, network=NETWORK, vocabulary=VOCAB_SIZE, max_len=MAX_LEN):
  '''Creates a LSTM, GRU or SRNN based model with a specified number of hidden units.
    args:
      layer_size: Number of hidden units in the LSTM, GRU or SRNN layer
      network: Name of network architecture
      vocabulary: Number of units in the Embedding layer
      max_len: Maximum length of a single input sequence
    returns:
      model: Trainable model'''
  model = Sequential()
  model.add(Embedding(vocabulary,vocabulary,input_length=max_len)) # Embeds the input sequence in the same dimensionality - simplifies the tf dataflow
  if network == 'LSTM':
    model.add(LSTM(layer_size))
  elif network == 'GRU':
    model.add(GRU(layer_size))
  elif network == 'SRNN':
    model.add(SimpleRNN(layer_size))
```

```
    else:
        raise ValueError("{} is not a valid network name. Valid network names are 'SRNN', 'LSTM' and 'GRU'.".format(NETWORK))
        return 0
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer=tf.optimizers.Adam(learning_rate=0.0001), metrics=['accuracy'])
    model.summary()

    return model

def train_model(X, Y, checkpoint_dir=CHECKPOINT_DIR):
    '''Trains the model on provided training data and saves the model weights for every epoch.
    Training stops either after the global maximum number of EPOCHS or if the the loss on the validation set val_loss does not improve for 3 epochs in a row, in which case the previously best model is used for the rest of the session.
      args:
        X: Preprocessed words
        Y: Target values
        hidden_units: Number of hidden units in the LSTM, GRU or SRNN layer
        corpus: Name of the training corpus
      returns:
        model: Trained model
    '''
    # Saving checkpoints: Directory, filenames, callback
    # Name of the checkpoint files, saving the values of loss, accuracy, val_loss and val_accuracy
    checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt_{epoch:02d}-{loss:.4f}-{accuracy:.4f}-{val_loss:.4f}-{val_accuracy:.4f}")
    #checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt_{epoch:02d}-{loss:.4f}-{accuracy:.4f}") # for len8 experiments
    checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_prefix, save_weights_only=True)

    # Create and train model
    model = build_model()
    model.fit(X,Y,batch_size=BATCH,epochs=EPOCHS,validation_split=0.15,
          callbacks=[EarlyStopping(monitor='val_loss', min_delta=0.0001, patience=3, restore_best_weights=True),
                  checkpoint_callback])

    return model

# ============ TRAINING/TESTING/EXPERIMENTS ============
def set_best_checkpoint(checkpoint_dir=CHECKPOINT_DIR):
```

```
  '''tf.train.latest_checkpoint() loads the latest saved model weights by referring to the file 'checkpoint' in the respective checkpoint directory.
  However, the latest model is not always the best.
  To circumvent this issue, this function overwrites the file 'checkpoint' to refer
  to the best (= lowest val_loss) model weight configuration, so tf.train.latest_checkpoint() instead retrieves the best model.
    args:
      checkpoint_dir: Directory whose checkpoint file is rewritten
    returns:
      None'''
  files = os.listdir(checkpoint_dir)
  weight_files = files[1:] # Ignore checkpoint file
  filenames = [filename[:-6] for filename in weight_files if filename.endswith('.index')]

  # Determine index of model weights with minimal val_loss
  val_losses = []
  for filename in filenames:
    prefix, train_loss, train_acc, val_loss, val_acc = filename.split('-')
    val_losses.append(float(val_loss))
  idx = val_losses.index(min(val_losses))
  best_weights = filenames[idx]

  # Overwrite checkpoint
  outfile_name = os.path.join(checkpoint_dir, 'checkpoint')
  outfile = open(outfile_name, 'w')
  outfile.write('model_checkpoint_path: "{0}"\nall_model_checkpoint_paths: "{0}"'.format(best_weights))
  outfile.close()

  return None

def train_test():
  '''Trains and tests the model with a train_test_split. Saves the accuracy and loss on the test data in a file with name '{NETWORK}_{HIDDEN_UNITS}.csv'.
  Overwrites 'checkpoint' file TF2.0 uses to store the name of the latest saved weights file with the name of the lowest val_loss weights file.
    args:
      none
    returns:
      none'''
  # Preprocessing
  X_train, X_test, Y_train, Y_test, tokenizer = prepare_training_data(
```

```
    CORPUS_DF)

  # Training model on train_split
  model = train_model(X_train, Y_train)

  # Testing on test_split
  test_accr = model.evaluate(X_test, Y_test) # test_accr = [loss,
    accuracy] on test data

  # Creating results file for test_split results
  test_dir = os.path.join('..', 'test_results', CORPUS)
  try:
    os.makedirs(test_dir)
  except FileExistsError:
    print("Directory {} already exists, proceeding with saving test
    results.".format(test_dir))
  test_out_filename = '{}_{}.csv'.format(NETWORK, HIDDEN_UNITS)
  test_out_path = os.path.join(test_dir, test_out_filename)

  # Saving test_split results
  test_out = open(test_out_path, 'w')
  test_out.write('Loss,Accuracy\n{:0.3f},{:0.3f}'.format(test_accr[0],
    test_accr[1]))
  test_out.close()

  # Set checkpoint file to refer to the best weights rather than the
    latest ones
  set_best_checkpoint()

def run_experiment(max_word_length=MAX_LEN, checkpoint_dir=
    CHECKPOINT_DIR):
  '''Makes the model classify experiment data specified in EXP_DF.
    Saves loss and accuracy in a '{NETWORK}_{HIDDEN_UNITS}.csv' file.
  Saves predictions per word in a 'detail_{NETWORK}_{HIDDEN_UNITS}.csv
     file in the shape of [preprocessed word], [predicted_value], [
    correct_value] for
  further analysis.
    args:
      max_word_length: Maximum wanted length of words
      corpus: Name of the training corpus
      network: Name of network architecture
      hidden_units: Number of hidden units in the LSTM, GRU or SRNN
    layer
    returns:
      None'''
  # Building model and assigning the weights of the model specified in
     its checkpoint file: model with the lowest val_loss
  model = build_model()
```

```
    model.load_weights(tf.train.latest_checkpoint(checkpoint_dir))

    # Creating tokenizer from training corpus to apply to experiment corpus
    discard_matrix, discard_matrix, discard_Y_train, discard_Y_test, tokenizer = prepare_training_data(CORPUS_DF, max_word_length)
    # Preprocessing experiment data
    exp_sequences_matrix, Y = prepare_experiment_data(EXP_DF, tokenizer)

    # Complete evaluation
    exp_accr = model.evaluate(exp_sequences_matrix,Y,batch_size=BATCH, verbose=0) # exp_accr = [loss, accuracy] on experiment data
    # Classifying experiment words on a word by word basis for detailed results
    exp = model.predict(exp_sequences_matrix)

    # Creating results file
    exp_dir = os.path.join('..', 'exp_results', CORPUS, EXPERIMENT)
    try:
        os.makedirs(exp_dir)
    except FileExistsError:
        print("Directory {} already exists, proceeding with experiment.".format(exp_dir))

    exp_out_filename = '{}_{}.csv'.format(NETWORK, HIDDEN_UNITS)
    exp_out_path = os.path.join(exp_dir, exp_out_filename)

    # Saving results
    exp_out = open(exp_out_path, 'w')
    exp_out.write('Loss,Accuracy\n{:0.3f},{:0.3f}'.format(exp_accr[0], exp_accr[1]))
    exp_out.close()

    # Creating detailed results file
    detail_out_filename = 'detail_{}_{}.csv'.format(NETWORK, HIDDEN_UNITS)
    detail_out_path = os.path.join(exp_dir, detail_out_filename)

    # Saving detailed results
    detail_out = open(detail_out_path, 'w')
    for i in range(len(exp)):
        detail_out.write('{},{},{}\n'.format(exp_sequences_matrix[i], exp[i][0], Y[i][0]))
    detail_out.close()

    return None

if MODE == 'train':
```

```
  train_test()
elif MODE == 'exp':
  run_experiment()
else:
  raise ValueError("{} is not an appropriate mode. Use 'train' or 'test' instead.".format(MODE))
```

Listing 3: Preprocessing input data, creating and training of RNN models. The trained models are then loaded to classify experiment data.

```python
# concat_results.py
# Creates a .csv overview over all model performances.
# Collecting calculated performance measures for all RNN models
import os
import pandas as pd

CORPORA = ['high','base','low']
EXPERIMENTS = ['ND','LRD']
OUTPATH_CSV = os.path.join('..', 'results')
OUTPATH_LATEX = os.path.join('..', 'latex', 'tab')
PERFORMANCE_COLUMNS = ['network', 'accuracy', 'precision', 'recall', 'f1_score', 'val_acc']

def read_checkpoint(ckpt_file):
    '''Reads 'checkpoint' file for epoch, train_loss, train_acc, val_loss and val_acc of the lowest val_loss model.
        args:
            ckpt_file: Path to the 'checkpoint' file
        returns:
            epoch: Epoch of the lowest val_loss model
            train_loss: Loss on training data of the lowest val_loss model
            train_acc: Accuracy on training data of the lowest val_loss model
            val_loss: Loss on validation data of the lowest val_loss model
            val_acc: Accuracy on validation data of the lowest val_loss model'''
    in_file = open(ckpt_file, 'r')
    lines = in_file.readlines()
    # Relevant field is in line 0 of checkpoint, second word and in quotation marks
    # Accessing relevant field and stripping quotation marks
    ckpt_string = lines[0].split()[1][1:-1] # "ckpt_epoch-train_loss-train_acc-val_loss-val_acc" without quotation marks
    ckpt_values = ckpt_string.split('_')[1]
    epoch = int(ckpt_values.split('-')[0])
    train_loss, train_acc, val_loss, val_acc = [float(value) for value in ckpt_values.split('-')[1:]]

    return epoch, train_loss, train_acc, val_loss, val_acc

def create_results(corpora=CORPORA, experiments=EXPERIMENTS):
    '''Collect all results across all corpora and experiments in a list of pd.DataFrames to create a dataframe containing all results. Used for data exploration and to create tables for the thesis.
            args:
                corpora: list of corpora to include in the big dataframe
                experiments: list of experiments to include in the big
```

```
    dataframe
      returns:
        results: dataframe with all specified results'''
  single_frames = []
  for corpus in corpora:
    for experiment in experiments:
      directory = os.path.join('..', 'exp_results', corpus, experiment)
      dirs = os.listdir(directory)
      for filename in dirs: # Going through results for all hidden_units configurations
        if not filename.startswith('detail_'): # Disregarding the detailed results
          network, hidden_units = filename[:-4].split('_')
          model_directory = os.path.join('..', 'saved_models', corpus, network, hidden_units)
          models = os.listdir(model_directory)
          ckpt = os.path.join(model_directory, models[0])
          epoch, train_loss, train_acc, val_loss, val_acc = read_checkpoint(ckpt)

          df = pd.read_csv(os.path.join(directory, filename),delimiter=',')
          df = df.loc[:, ~df.columns.str.contains('^Unnamed')]
          df = df.rename(str.lower, axis='columns')

          # Putting all measured values for this model together
          df['val_acc'], df['val_loss'], df['train_acc'], df['train_loss'], df['epoch'], df['network'], df['experiment'], df['corpus'], df['hidden_units']  = val_acc, val_loss, train_acc, train_loss, epoch, network, experiment, corpus, hidden_units
          col_order = ['network', 'hidden_units', 'corpus', 'experiment', 'accuracy', 'precision', 'recall', 'f1_score', 'epoch', 'train_acc', 'train_loss', 'val_acc', 'val_loss']
          df = df[col_order]
          single_frames.append(df)
  # Join all small dfs into the complete overview of all results
  results = pd.concat(single_frames, ignore_index=True)
  results = results.sort_values(by='accuracy', ascending=False)

  return results

def save2file(dataframe, experiment, corpus, mode, outpath):
  '''Saves a given dataframe as either a .csv or .tex table. Latex formatting was then manually modified.
      args:
        dataframe: dataframe containing measurements
        experiment: string, name of the current experiment
```

```
      corpus: string, name of the current corpus
      mode: string, csv/latex, determines which file to generate
      outpath: string, determines where the file is saved_models
    returns:
      none'''
  try:
    os.makedirs(outpath)
  except:
    print("Directory {} exists, proceeding.".format(outpath))
  if mode=='csv':
    dataframe.to_csv(os.path.join(outpath, 'results_{}_{}.csv'.format(
   experiment, corpus)), index=False)
  elif mode=='latex': # Latex output has been used as a basis for
   tables in the thesis, heavy modifications were made
    dataframe.to_latex(os.path.join(outpath, 'results_{}_{}.tex'.
   format(experiment, corpus)), columns=PERFORMANCE_COLUMNS,
   float_format='{:0.3f}'.format, index=False)

def create_all_tables(df, mode):
  '''Creates tables collecting all results given the supplied mode.
    args:
      df: dataframe containing measurements
      mode: string, LRD/ND/SRNN/LSTM/GRU, determines what to filter
   the dataframe for
    returns:
      none'''
  corpora = ['base', 'low', 'high']
  measures = []
  if mode in ('SRNN', 'LSTM', 'GRU'):
    data = df.loc[df.network == mode]
    for corpus in corpora:
      data_corpus = data.loc[data.corpus == corpus]
      save2file(data_corpus, mode, corpus, 'csv', OUTPATH_CSV)
      print(mode, corpus)
      mean = data_corpus.mean()
      std = data_corpus.std()
      total = pd.concat([mean, std], axis=1)
      print(total.T)
      measures.append(total.T)
    overview = pd.concat(measures)
    print(overview)
  elif mode in ('LRD', 'ND'):
    data = df.loc[df.experiment == mode]
    for corpus in corpora:
      data_corpus = data.loc[data.corpus == corpus]
      save2file(data_corpus, mode, corpus, 'csv', OUTPATH_CSV)

results = create_results()
```

```
results['hidden_units'] = results['hidden_units'].astype(int)
results = results.sort_values(by=['network', 'hidden_units'],
    ascending=True)

create_all_tables(results, 'LRD')
create_all_tables(results, 'ND')
create_all_tables(results, 'SRNN')
create_all_tables(results, 'LSTM')
create_all_tables(results, 'GRU')
```

Listing 4: Script to create the basis of the tables featured in Section 4.1.

```python
# analyze_performance.py
# Functions to prepare network predictions for further analysis.
# Creates dataframes collecting the predictions of all outlier networks for error analysis.
# Only false positives were discussed in this thesis.
# To be used as 'python analyze_performance.py > some_outfile.csv'

import os
import pandas as pd
import corpus_tools
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
import numpy as np

def numbers_to_words(corpus, experiment, network, hidden_units):
  '''Cleans up the saved detailed RNN predictions by removing additional characters and
  translating character indices back to legible strings.
      args:
        corpus: string, name of the corpus
        experiment: string, name of the experiment
        network: string, name of the network architecture
        hidden_units: int, number of hidden units
      returns:
        data_matrix: list of clean data, ready to be turned into a pd.DataFrame'''
  detail_file = os.path.join('..', 'exp_results', corpus, experiment, 'detail_{}_{}.csv'.format(network, hidden_units))
  training_df = pd.read_csv('../training/{}.csv'.format(corpus), delimiter=',',encoding='latin-1')

  table = str.maketrans(dict.fromkeys(' []')) # Remove superfluous characters
  lines = []
  with open(detail_file, 'r') as f:
    for line in f:
      line = line.translate(table).strip().split(',')
      lines.append(line)
  prev_line = []
  mod_lines = []
  # Merge lines split by \n in saving
  for i in range(2,len(lines), 2):
    line = lines[i]
    prev_line = lines[i-1]
    prev_line.extend(line)
```

```
    mod_lines.append(prev_line)
  # Decode numbers to characters
  if experiment == 'LRD':
    num_to_char = {'1': ']', '2': '[', '3': '{', '4': '}', '5': '$'}
  else:
    num_to_char = {'1': ']', '2': '{', '3': '[', '4': '}', '5': '$'}
  # Prepare matrix to be turned into a df
  data_matrix = [[line[0].translate(line[0].maketrans(num_to_char))+
   line[1].translate(line[1].maketrans(num_to_char)), float(line[2]),
    int(line[3])] for line in mod_lines]

  return data_matrix

def prepare_dataframe(corpus, experiment, network, hidden_units,
   GLOBAL_WORDS):
  '''Processes the data_matrix from numbers_to_words into a pd.
   DataFrame for further analysis.
      args:
        detail_file: Path to a prediction-per-word file
      returns:
        df: pd.DataFrame with all columns relevant for analysis'''
  data_matrix = numbers_to_words(corpus, experiment, network,
   hidden_units)
  df = pd.DataFrame(data_matrix, columns=['words', 'predictions', '
   golds'])
  df['words'] = GLOBAL_WORDS
  # Append columns determining the
  # - kind of error in the word if the word is wrong
  # - position of the corrupted character if it can be determined
  # - nesting depth at error position
  # - running bracket distance at error position
  df['max_valid_nesting_depth'] = df.words.map(lambda word:
   corpus_tools.maxValidNestingDepth(word))
  df['error'] = df.words.map(lambda word: corpus_tools.determineError(
   word))
  df['error_pos'] = df.words.loc[df.error != 'none'].map(lambda word:
   corpus_tools.findErrorPosition(word))
  df['error_depth'] = df.words.loc[df.error != 'none'].map(lambda word
   : corpus_tools.nestingDepthAtPosition(word, corpus_tools.
   findErrorPosition(word)))
  df['error_distance'] = df.words.loc[df.error != 'none'].map(lambda
   word: corpus_tools.bracketDistanceAtPosition(word, corpus_tools.
   findErrorPosition(word)))
  #print(df.head())
  return df

def measure_performance(results):
  '''Calculates precision, recall and F1 score of a dataframe
```

```python
  containing predictions and gold labels.
     args:
       results: Dataframe containing predictions and gold labels
     returns:
       precision: TPs/(TPs+FPs)
       recall: TPs/(TPs+FNs)
       f1_score: 2*((precision*recall)/(precision+recall))'''
  true_pos = len(results.loc[results.golds == 1].loc[results.predictions >= 0.5])
  false_pos = len(results.loc[results.golds == 0].loc[results.predictions >= 0.5])
  true_neg = len(results.loc[results.golds == 0].loc[results.predictions < 0.5])
  false_neg = len(results.loc[results.golds == 1].loc[results.predictions < 0.5])
  if true_pos+false_pos == 0:
    precision = 0.
    recall = 0.
    f1_score = 0.
  else:
    precision = float(true_pos/(true_pos+false_pos))
    recall = float(true_pos/(true_pos+false_neg))
    if precision+recall == 0.:
      f1_score = 0.
    else:
      f1_score = 2.*((precision*recall)/(precision+recall))
  return precision, recall, f1_score

def extend_performance_measures():
  '''Iterates through all files containing experiment results (accuracy, loss) and their corresponding detail files, which contain every single word, prediction and gold label. From that, precision, recall and F1 score are calculated and added to the experiment results file.
     args:
       none
     returns:
       none'''
  corpora = ['base', 'high', 'low']
  experiments = ['LRD', 'ND']
  networks = ['SRNN', 'GRU', 'LSTM']
  for corpus in corpora:
    for experiment in experiments:
      for network in networks:
        for hidden_units in [2**i for i in range(1,10)]:
          # Appending precision, recall, f1_score to sparse_file
          detail_file = os.path.join('..', 'exp_results', corpus, experiment, 'detail_{}_{}.csv'.format(network, str(hidden_units)))
```

```python
        sparse_file = os.path.join('..', 'exp_results', corpus, experiment, '{}_{}.csv'.format(network, str(hidden_units)))
        sparse = pd.read_csv(sparse_file)
        # Processing values in the details file to calculate additional performance measures
        details = numbers_to_words(detail_file)
        precision, recall, f1_score = measure_performance(details)
        sparse['precision'], sparse['recall'], sparse['f1_score'] = precision, recall, f1_score
        sparse.to_csv(sparse_file, index=False)
        # Saving the extended values
        prepend_header(detail_file)

def breakdownPredictions(df, corpus, experiment, network, hidden_units, performance):
  '''Transforms a full dataframe of word-prediction-gold_label data into 4 dataframes specific to a single model: true positives, false positives, true negatives and false negatives.
    args:
      df: Complete dataframe of word-prediction-gold_labels
      corpus, experiment, network, hidden_units, performance: Strings filtering the df for the specific model
    returns:
      true_positives, false_positives, true_negatives, false_negatives: Dataframes containing all TPs, FPs, TNs and FNs of a specific model'''
  generals = pd.DataFrame()
  generals['corpus'], generals['experiment'], generals['network'], generals['hidden_units'], generals['performance'] =  pd.Series(corpus), experiment, network, hidden_units, performance
  # Determine all predictions in their own series to analyze.
  true_positives = df.loc[df.golds == 1].loc[df.predictions >= 0.5]
  false_positives = df.loc[df.golds == 0].loc[df.predictions >= 0.5]
  true_negatives = df.loc[df.golds == 0].loc[df.predictions <= 0.5]
  false_negatives = df.loc[df.golds == 1].loc[df.predictions <= 0.5]

  true_positives = true_positives.join(generals)
  false_positives = false_positives.join(generals)
  true_negatives = true_negatives.join(generals)
  false_negatives = false_negatives.join(generals)

  for column in generals.columns:
    true_positives[column] = true_positives[column].fillna(generals[column][0])
    false_positives[column] = false_positives[column].fillna(generals[column][0])
    true_negatives[column] = true_negatives[column].fillna(generals[column][0])
```

```
    false_negatives[column] = false_negatives[column].fillna(generals[
  column][0])

  return true_positives, false_positives, true_negatives,
  false_negatives

def prepend_header(detail_file):
  '''Includes a descriptive header in a file containing individual
  words, predictions and their gold label.
    args:
      detail_file: Path to a file containing individual words,
  predictions and their gold label
    returns:
      none'''
  f = open(detail_file,'r')
  temp = f.read()
  f.close()

  f = open(detail_file, 'w')
  f.write('words,predictions,golds\n')

  f.write(temp)
  f.close()

def create_mega_df():
  '''Creates and saves four dataframes containing every single word,
  prediction and gold label for every outlier model (accuracy
  >55%/<45%), split into true positives, false positives, true
  negatives and false negatives.'''
  global_words_LRD = numbers_to_words('base', 'LRD', 'LSTM', '8')
  df = pd.DataFrame(global_words_LRD, columns=['words', 'predictions',
    'golds'])
  GLOBAL_WORDS_LRD = df['words']
  global_words_ND = numbers_to_words('base', 'ND', 'SRNN', '2')
  df = pd.DataFrame(global_words_ND, columns=['words', 'predictions',
  'golds'])
  GLOBAL_WORDS_ND = df['words']

  true_positives_frames = []
  false_positives_frames = []
  true_negatives_frames = []
  false_negatives_frames = []
  valid_nesting_depth_frames = []
  error_pos_frames = []
  error_depth_frames = []
  networks = ['base LRD LSTM 8 good', 'base LRD GRU 2 good', 'base LRD
    GRU 128 good',
    'base LRD GRU 32 bad', 'base LRD LSTM 128 bad', 'base LRD LSTM 16
```

```
    bad', 'base LRD SRNN 16 bad',
    'low LRD GRU 2 good', 'low LRD SRNN 4 good', 'low LRD LSTM 16 good', 'low LRD GRU 64 good',
    'low LRD LSTM 4 bad',
    'high LRD GRU 512 good',
    'high LRD LSTM 8 bad', 'high LRD GRU 8 bad', 'high LRD LSTM 4 bad',
    'base ND LSTM 128 good', 'base ND LSTM 32 good', 'base ND GRU 512 good', 'base ND SRNN 2 good', 'base ND GRU 4 good',
    'base ND SRNN 128 bad', 'base ND SRNN 256 bad',
    'low ND LSTM 512 good', 'low ND GRU 64 good', 'low ND SRNN 32 good', 'low ND LSTM 16 good', 'low ND GRU 4 good', 'low ND LSTM 8 good',
    'low ND SRNN 4 bad', 'low ND LSTM 32 bad',
    'high ND SRNN 16 good',
    'high ND LSTM 64 bad']

for entry in networks:
  corpus, experiment, network, hidden_units, performance = entry.split()
  # Set translation table for numbers_to_words
  if experiment == 'LRD':
    GLOBAL_WORDS = GLOBAL_WORDS_LRD
  else:
    GLOBAL_WORDS = GLOBAL_WORDS_ND
  details = prepare_dataframe(corpus, experiment, network, hidden_units, GLOBAL_WORDS)
  true_positives, false_positives, true_negatives, false_negatives = breakdownPredictions(details, corpus, experiment, network, hidden_units, performance)
  true_positives_frames.append(true_positives)
  false_positives_frames.append(false_positives)
  true_negatives_frames.append(true_negatives)
  false_negatives_frames.append(false_negatives)
# Concatenate all results
mega_true_positives = pd.concat(true_positives_frames)
mega_false_positives = pd.concat(false_positives_frames)
mega_true_negatives = pd.concat(true_negatives_frames)
mega_false_negatives = pd.concat(false_negatives_frames)

# Save results
tp_out = os.path.join('..','results','mega_true_positives.csv')
fp_out = os.path.join('..','results','mega_false_positives.csv')
tn_out = os.path.join('..','results','mega_true_negatives.csv')
fn_out = os.path.join('..','results','mega_false_negatives.csv')
mega_true_positives.to_csv(tp_out)
mega_false_positives.to_csv(fp_out)
mega_true_negatives.to_csv(tn_out)
```

```
    mega_false_negatives.to_csv(fn_out)

def count_error_types(fp, experiment, network, corpus, performance):
    '''Determines number of open/closed bracket error words in a dataframe.
        args:
          fp: dataframe containing either false positives or true negatives (since other categories do not have incorrect words)
          experiment: string, name of the experiment
          network: string, name of the network architecture
          hidden_units: int, number of hidden units
        returns:
          none'''
    open = fp.loc[fp.experiment == experiment].loc[fp.network == network].loc[fp.corpus == corpus].loc[fp.performance == performance].loc[fp.error == 'open'].count()[1]
    closed = fp.loc[fp.experiment == experiment].loc[fp.network == network].loc[fp.corpus == corpus].loc[fp.performance == performance].loc[fp.error == 'closed'].count()[1]
    total = open + closed
    if total:
      if closed:
        ratio = open/closed
      else:
        ratio = np.inf

      print("{},{},{},{},{},{},{},{}".format(network, experiment, corpus.capitalize(), performance, open, closed, total, ratio))

def print_bracket_ratio_table():
    '''Creates a table splitting false positives into open/closed error types for each outlier network, sorted by 'good' and 'bad' outliers.'''
    fp_in = os.path.join('..','results','mega_false_positives.csv')
    fp = pd.read_csv(fp_in)
    experiments = ['LRD', 'ND']
    networks = ['SRNN', 'LSTM', 'GRU']
    corpora = ['base', 'low', 'high']
    print("network,experiment,corpus,performance,open,closed,total,ratio")
    for experiment in experiments:
      for corpus in corpora:
        for network in networks:
          count_error_types(fp, experiment, network, corpus, 'good')
    for experiment in experiments:
      for corpus in corpora:
        for network in networks:
          count_error_types(fp, experiment, network, corpus, 'bad')
```

```
create_mega_df()
print_bracket_ratio_table()
```

Listing 5: Script to create the basis of the tables featured in Section 4.4.

Eidesstattliche Erklärung

Eidesstattliche Erklärung zur Bachelorarbeit

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Unterschrift Ort, Datum