Closing Brackets with Recurrent Neural Networks

Natalia Skachkova[*], **Thomas A. Trost**[*], **Adam Kusmirek** and **Dietrich Klakow**
Spoken Language Systems
Saarland University
Saarland Informatics Campus
66123 Saarbrücken, Germany
{s9naskac@stud,{thomas.trost,akusmirek,dietrich.klakow}@lsv}.uni-saarland.de

Abstract

Many natural and formal languages contain words or symbols that require a matching counterpart for making an expression well-formed. The combination of opening and closing brackets is a typical example of such a construction. Due to their commonness, the ability to follow such rules is important for language modeling. Currently, recurrent neural networks (RNNs) are extensively used for this task. We investigate whether they are capable of learning the rules of opening and closing brackets by applying them to synthetic Dyck languages that consist of different types of brackets. We provide an analysis of the statistical properties of these languages as a baseline and show strengths and limits of Elman-RNNs, GRUs and LSTMs in experiments on random samples of these languages. In terms of perplexity and prediction accuracy, the RNNs get close to the theoretical baseline in most cases.

1 Introduction

Brackets are a challenge for language models. They regularly appear in texts, they typically produce long-range dependencies, and a failure to properly close them is readily recognized by a human evaluator as a severe error (Shen et al., 2017). Beyond the syntactical level, many natural languages exhibit brackets-like structures. For example, the German language is infamous for its convoluted sentences with verb-particle constructions, in which words from the beginning have to be properly closed at the end (Dewell, 2011; Müller et al., 2015).

In this paper we present a dedicated study of the capability of Elman-RNNs, GRUs and LSTMs to model expressions with brackets and properly close them. Towards this end, we conduct experiments on Dyck languages, which consist of balanced bracket expressions.

1.1 Related Work

Synthetic datasets and formal languages have long been used for checking the ability of RNNs to capture a particular feature. For example, Elman (1990), Das et al. (1992), or Gers and Schmidhuber (2001) already did such investigations. Recent work in this direction was done by Weiss et al. (2017, 2018).

More specifically, the interplay of RNNs with certain grammatical constructs, brackets and Dyck languages has been the subject of several studies. Karpathy et al. (2016) show that RNNs are capable of capturing bracket structures on real-world datasets. Linzen et al. (2016) study the application of LSTMs to certain grammatical phenomena. RNNs and their variants have been used for recognizing Dyck words (Kalinke and Lehmann, 1998; Deleu and Dureau, 2016). Li et al. (2017) evaluate their nonlinear weighted finite automata model on a Dyck language. Most recently, Bernardy (2018) conducted a very similar study to ours on Dyck languages with a slightly different focus.

1.2 Contributions

In this work, we sample Dyck words in such a way that we can give theoretical lower bounds for the perplexity of a respective language model. This way, we can compare the performance of RNNs with a theoretical baseline and not just with the performance of other architectures.

2 Dyck Languages

We use artificially generated data in order to have a completely controlled environment for the experiments. In particular, the training and test datasets consist of balanced sequences of n different types

[*] Both authors contributed equally.

of brackets, $(_1,)_1, (_2,)_2, \ldots$, where n depends on the specific experiment. The set of such sequences forms the so-called *Dyck language* D_n (Duchon, 2000). Elements of D_n are called *Dyck words*. Each D_n is a context-free but not regular formal language (Chomsky, 1956) and can be described by the grammar:

$$S \to \epsilon \tag{1a}$$

$$S \to SS \tag{1b}$$

$$S \to (_i S)_i \quad \forall i \in \{1, \ldots, n\} \tag{1c}$$

Some examples of such Dyck words are:

$$(_1 \, (_1 \,)_1 \,)_1$$
$$(_2 \,)_2 \, (_1 \,)_1 \, (_3 \, (_1 \,)_1 \,)_3$$
$$(_1 \, (_1 \,)_1 \, (_1 \,)_1 \, (_1 \,)_1 \, (_1 \,)_1 \,)_1$$

It is well-known that there are $C_m := \frac{1}{m+1}\binom{2m}{m}$ words of length $2m$ in D_1, where C_m is the m-th *Catalan number* (Chung and Feller, 1949). As the type of each pair of brackets can independently be chosen, it follows that there are $n^m C_m$ words of length $2m$ in D_n. There are obviously no Dyck words with odd length.

2.1 Generation of Dyck Words

Each "sentence" in the datasets is a randomly generated non-empty Dyck word. The first symbol of a word is always an open bracket. From there, the generation proceeds in a sequential manner: With probability p an open bracket is emitted. Otherwise and thus with probability $p - 1$, a matching closed bracket is emitted or the generation terminates if all open brackets already have a matching partner. If not stated otherwise, we assume $0 < p < 1$ in all calculations because the edge cases usually have to be treated differently but do not add significant value to our discussion. The type of bracket is chosen randomly from a uniform distribution but might follow some other distribution for future studies.

2.2 Statistical Properties of Dyck Words

We quickly review some statistical properties of such sequences for explaining choices in the setups and in order to get a baseline for the experiments. It can readily be seen that the sequences of length $2m$ all have the same probability:

$$\frac{1}{n^m} p^{m-1} (1-p)^{m+1} \tag{2}$$

The asymmetry in the exponents is due to leaving out empty sequences. The factor n^{-m} accounts for the equally probable choices of brackets types. We can check consistency by considering the normalization condition:

$$\sum_{k=1}^{\infty} C_k p^{k-1} (1-p)^{k+1} = \begin{cases} 1 & \text{for } p \leq \frac{1}{2} \\ \frac{(1-p)^2}{p^2} & \text{for } p > \frac{1}{2} \end{cases} \tag{3}$$

While the result for $p \leq \frac{1}{2}$ is expected, the case $p > \frac{1}{2}$ might appear curious at first. The reason for this behaviour is that the sum only takes finite sequences into account, while there is a non-zero probability for getting infinite sequences for $p > \frac{1}{2}$. This is easily seen for the case $p = 1$, where brackets are never closed so that the overall probability of obtaining a finite sequence is indeed zero.

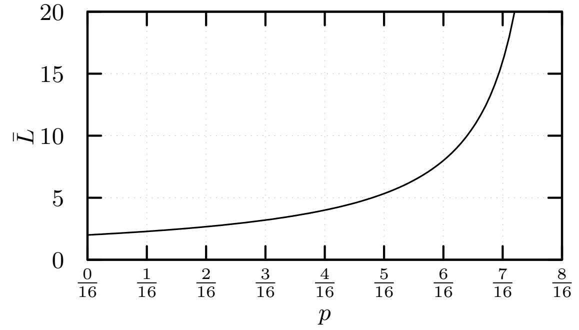


Figure 1: Average length of generated sequences, based on Eq. (4).

2.2.1 Average Length

This naturally leads to the question what the average length $\bar{L}$ of the sequences is, depending on p. For $p < \frac{1}{2}$, we find

$$\bar{L} = 2 \sum_{k=1}^{\infty} k C_k p^{k-1} (1-p)^{k+1} = \frac{2}{1-2p}. \tag{4}$$

The graph of this function can be seen in Fig. 1. In line with our previous findings, problems with infinite sequences arise for $p \geq \frac{1}{2}$, as the expression (4) diverges for $p = \frac{1}{2}$. For these reasons, we only consider the case $p < \frac{1}{2}$ in the experiments.

2.2.2 Baseline for Perplexity

A prediction system for the next symbol emitted by the generator will not be able to perform arbitrarily well due to the random nature of the process. In order to get a baseline for the performance, we consider the perplexity per symbol PP of the probability distribution of the generated Dyck languages. For a sequence of symbols $\boldsymbol{w}$

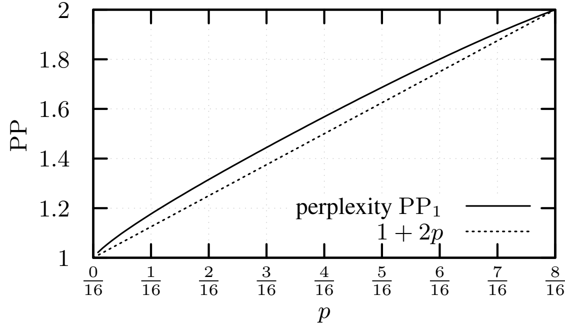


Figure 2: Perplexity PP_1 of distribution of D_1, cf. (7). As a reference, the graph of $1+2p$ is given.

with length $|\boldsymbol{w}|$ it is defined as

$$\mathrm{PP} := 2^{-\frac{1}{|\boldsymbol{w}|}\sum_{i=1}^{|\boldsymbol{w}|}\log_2 P(w_i|w_1,\ldots,w_{i-1})}, \quad (5)$$

where $P(w_i|w_1,...,w_{i-1})$ is the probability of the i-th symbol under the model, given the previous $i-1$ symbols. Eq. (5) corresponds to the way in which perplexity is calculated by the software that we use for our experiments (cf. Sec. 3).

We estimate the baseline for the perplexity PP_n for the language D_n by considering (5) under the true probability model in the limit of an infinite amount of samples from the corresponding probability distribution. Under these conditions and for our case, (5) can be transformed into:

$$\log_2 \mathrm{PP}_n = -\lim_{L\to\infty}\lim_{N\to\infty}\frac{\sum_{\ell=1}^{L}\frac{N_\ell}{N}\log_2 P_\ell}{\sum_{\ell=1}^{L}\frac{N_\ell}{N}(2\ell+1)} \quad (6)$$

The numerator contains the sum of the log-probabilities, where P_ℓ is the probability of a Dyck word with 2ℓ brackets. The denominator represents the total number of symbols. Adding one to the length in the term $(2\ell+1)$ accounts for the end-of-sentence symbols that are counted by the software. The limit $L\to\infty$ for the maximum length of a word is taken at the end because the normalization by the number of symbols has to be carried out for a finite value. Finally, N represents the number of samples and N_ℓ stands for the number of words with 2ℓ brackets in the dataset, so that $\frac{N_\ell}{N}$ converges to the probability of generating a word of this length.

All these quantities are known, so that we can obtain the following result:

$$\mathrm{PP}_n = n^{\frac{1}{3-2p}}\frac{1}{\sqrt{p(1-p)}}\left(\frac{p^3}{1-p}\right)^{\frac{1-2p}{6-4p}} \quad (7)$$

While the expression (7) with its singularity at zero does not readily reveal the characteristics of the perplexity, its graph (Fig. 2) shows that it is close to a simple affine function. In the edge cases it behaves just like expected: $p=0$ means that there is no randomness at all and there is just one possible next symbol. For $p=\frac{1}{2}$ however, opening and closing brackets are equally likely, so that there are always two symbols to choose from without any way to tell which to prefer. The dependence on n must be sublinear because for closing brackets the type is uniquely predictable.

3 Neural Network Architecture

We use three different RNN architectures for our experiments: Elman-RNN (abbreviated as SRNN for *simple* RNN), GRU (*gated recurrent unit*), and LSTM (*long short-term memory*).

For the experiments with SRNNs we use the RNNLM toolkit (version 0.3e) developed by Mikolov et al. (2011b). The SRNN has one hidden layer of arbitrary size N_{hidden} with a sigmoid activation function. At each time step the input vector is built by concatenating the vector of the current word and the output produced by the hidden layer during the previous time step. The next word is predicted by applying the softmax function to the last layer. The RNNLM toolkit offers the possibility to group words into classes (Mikolov et al., 2011a), but this feature is more interesting for boosting efficiency in cases of large vocabularies with a natural frequency distribution. After initializing the weights with random Gaussian noise, the training of the SRNNs is performed using the standard stochastic gradient descent algorithm with an initial learning rate $\alpha=0.1$ and the recurrent weight is trained by the truncated backpropagation through time algorithm (Rumelhart et al., 1985). We refer to the respective hyperparameter that specifies the number of time steps taken into account as T_{BPTT}.

For the other more elaborate architectures, we make use of TF-NNLM-TK[1] by Oualil et al. (2016) which provides implementations of LSTM and GRU networks. LSTMs and GRUs work similar to SRNNs but exhibit specific units that allow for storing previous activations and for tracking long-term dependencies in a more flexible and efficient way. In the case of LSTMs, the memory state is being handled via input, forget and output

[1] https://github.com/uds-lsv/TF-NNLM-TK

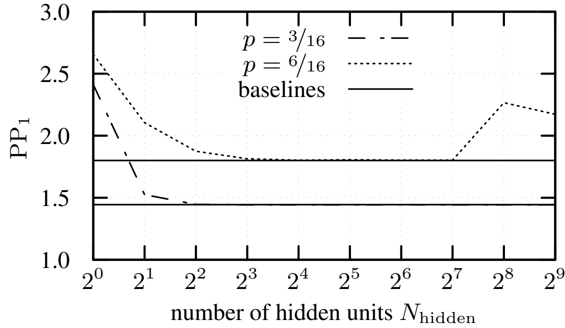


Figure 3: Test perplexity PP_1 vs. number of hidden units N_{hidden} for SRNN. Both curves reach their respective baseline, given in (7).

gate. Those gates allow to decide on the amount of a cell state that should be preserved or forgotten and the amount that should be passed to the cells in the next layer of the network. Similarly, the GRU regulates its memory state using an update and a reset gate that allows to either delete the previous cell state and decide on the amount of the current activation that should be added to the current cell state. For the experiments with LSTMs and GRUs the hyperparameter settings are chosen to be similar to the ones used with the RNNLM toolkit. Weights are again initialized using random Gaussian noise and standard stochastic gradient descent is utilized. The initial learning rate is set to $\alpha = 0.1$ and in later training epochs decayed using a factor of $\gamma = 0.9$. The models are trained for 100 epochs using a batch size of 128 and $T_{\text{BPTT}} = 16$ (if not stated otherwise) with learning rate decay starting at epoch 80.

4 Experiments

4.1 Setup and Perplexities

We conduct a number of experiments for investigating the overall performance and the influence of the hyperparameters on the perplexity. For all experiments we use datasets that were artificially generated in the previously described way (cf. Sec 2.1). All training sets contain 131,072 Dyck words, while the test sets contain 10,000 Dyck words that were sampled from the same distribution. In all experiments, the value of p is varied between $1/16$ and $7/16$ in steps of $1/16$. The ratio behind this choice is that $7/16$ yields an average sequence length of 16, which is roughly a typical sentence length for natural languages (Sichel, 1974; Sigurd et al., 2004). The smaller values of p are considered for comparison.

n	1	2	3	4	5
baseline	1.444	1.881	2.195	2.449	2.667
GRU	1.450	1.900	2.204	2.488	2.691
LSTM	1.451	1.899	2.203	2.486	2.688
RNN	1.445	1.873	2.205	2.445	2.669

(a) For $p = 3/16$.

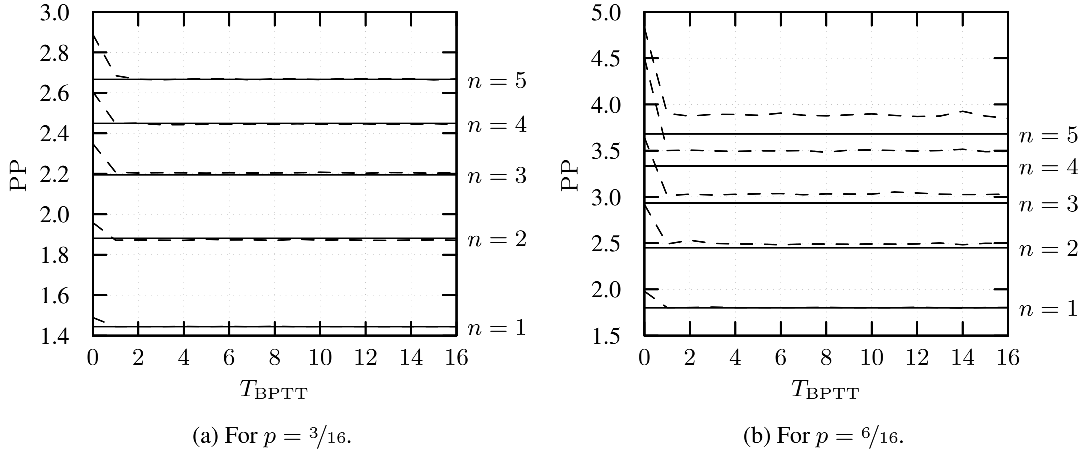
n	1	2	3	4	5
baseline	1.800	2.450	2.934	3.334	3.682
GRU	1.808	2.483	2.995	3.396	3.775
LSTM	1.810	2.481	2.995	3.401	3.771
RNN	1.804	2.494	3.030	3.499	3.885

(b) For $p = 6/16$.

Table 1: Baseline respectively mean test perplexity PP_n for T_{BPTT} settings between 1 and 16 in steps of 1 for different architectures (cf. Fig. 4 for a graphical representation of the SRNN values). The standard deviation is roughly around 0.001 and slightly larger for the SRNN.

In a first set of experiments, we consider D_1 and vary the number of hidden units between 1 and 512, doubling the hidden layer size in each iteration. Having more than 512 units does not bring much perplexity improvement but slows down the training process considerably. Typical results for the Elman-RNN can be found in Fig. 3. For all values of p, the test time perplexity reaches the baseline. The convergence is slower for larger values of p, which is the expected behavior. For larger values of N_{hidden} there are some increases of PP_1 that are most probably connected to the specific software implementation. Despite that, the models are surprisingly good at recovering the baseline. All in all we conclude that $N_{\text{hidden}} = 64$ is a good compromise between optimization for perplexity and speed.

In a second set of experiments, we change T_{BPTT} from 0 to 16, increasing its size by one in each iteration. We limit T_{BPTT} with 16 as this is the maximum expected length of a sentence in our setting. This time, we do not only vary p, but also the number of types of brackets n. Typical results can be seen in Fig. 4. It is striking that T_{BPTT} has hardly any influence on the performance as long as it is larger than zero. This can be exploted for making the comparison easier: The average value of the test perplexities for the different architectures is given in Tab. 1. The values give a good impression of where the curves saturate. Higher values of p and n appear to make the task more



(a) For $p = 3/16$. (b) For $p = 6/16$.

Figure 4: Test perplexity PP_n vs. hyperparameter T_{BPTT} for Dyck languages with different numbers of types of brackets, obtained with the Elman-RNN. The respective baselines (cf. Eq. (7)) are plotted as solid lines.

p	n	$\bar{L}$	$\bar{\ell}$	SRNN	GRU	LSTM
$3/16$	1	3.191	2.586	1.00	1.000	1.000
$3/16$	2	3.222	2.611	0.978	0.9998	1.000
$3/16$	4	3.183	2.609	0.960	0.9994	1.000
$6/16$	1	8.049	4.976	1.00	0.9997	0.9999
$6/16$	2	7.941	4.936	0.742	0.9982	0.9996
$6/16$	4	8.095	5.079	0.966	0.9959	0.9998

Table 2: Accuracy for the task of finding the last bracket of a Dyck word, together with measured values for the average length $\bar{L}$ of the words and the average length of the task $\bar{\ell}$ (see the text for a definition).

challenging. While all curves rsp. values are close to the baseline in Fig. 4a and Tab. 1a, the gap increases with n in Fig. 4b and Tab. 1b. Given that the average length of Dyck words for $p = 3/16$ is only 3.2, compared to a length of 8 for $p = 6/16$, the differences in the performance is not surprising. While the Elman-RNN performs similar or even slightly better than the other architectures for the easier tasks, the more elaborate methods increasingly outperform it with increasing task difficulty.

4.2 Accuracy

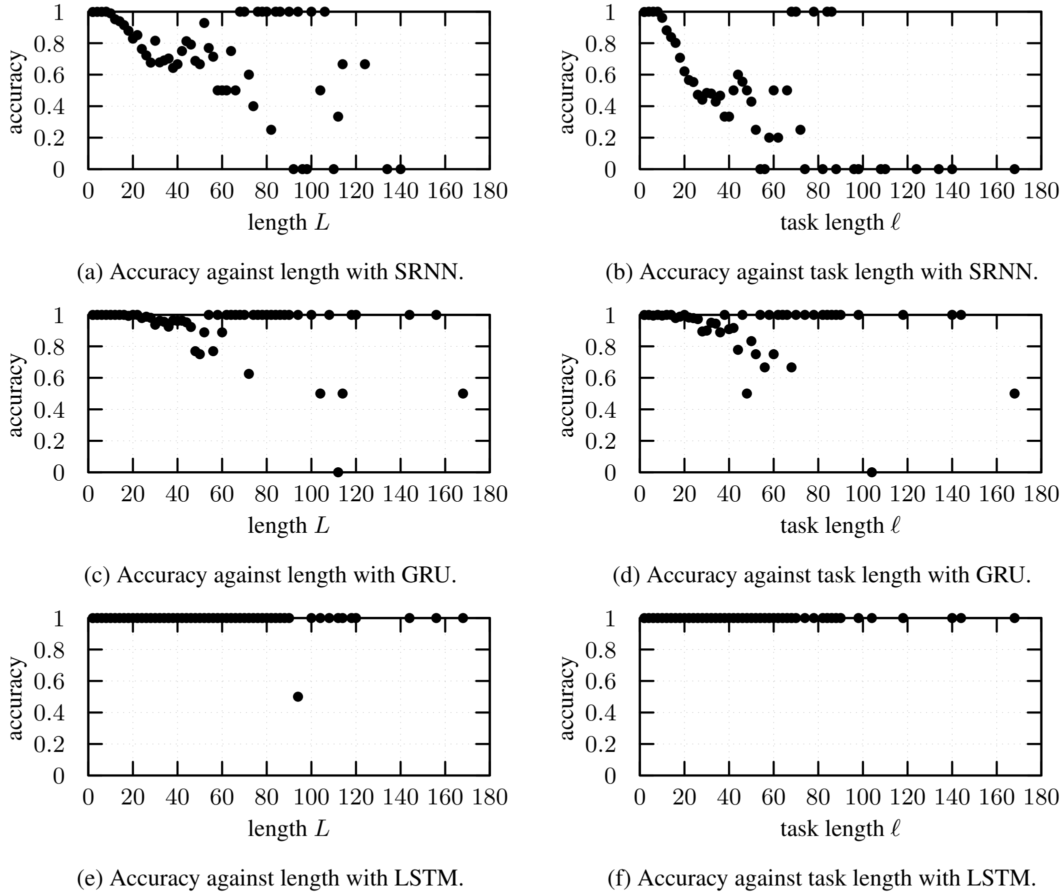
Based on the results of the previous section, the RNNs appear to perform quite well in terms of the perplexity. In order to get a better feeling for the capability of the networks, we consider a second task: Given a Dyck word without the last closing bracket, the RNN has to predict the most likely candidate for this missing symbol. The success is measured in terms of accuracy, which is the number of successfully finished tasks divided by the total number of tasks. The respective values, based on a dataset of 10000 Dyck words, are given in Tab. 2. Except for one case, the RNNs reach an accuracy close to one. Only one experiment is done per configuration and even harder tasks appear to be solvable, so the lower value is probably just an outlier. GRU and in particular LSTM perform almost perfectly in this task.

Some additional statistics are given in the table. The average word length $\bar{L}$ indeed follows (4). Besides that, a new quantity is introduced here: The average length of the task $\bar{\ell}$ measures how far the algorithm has to look back in order to find the open brackets it has to close on average. The difference between length L and task length ℓ is best illustrated with an example. For the Dyck word

$$(_2\ (_1\ )_1\ )_2\ \underbrace{(_1\ (_1\ )_1\ (_3\ )_3\ \boxed{)_1}}_{\text{task length } \ell = 6},$$

the length is ten but the task length is six because the first four brackets are irrelevant for determining the last one, which is boxed for emphasis. The task length is the relevant measure for the hardness of the task, because small values of $\bar{\ell}$ would mean that there are hardly any long-range dependencies. For $p = 6/16$, $\bar{\ell}$ lies around five, so we would expect to need at least a five-gram model or something equivalent for achieving good results in this task.

The full frequency distribution of length and task length can be seen in Fig. 6. By far the largest



(a) Accuracy against length with SRNN.

(b) Accuracy against task length with SRNN.

(c) Accuracy against length with GRU.

(d) Accuracy against task length with GRU.

(e) Accuracy against length with LSTM.

(f) Accuracy against task length with LSTM.

Figure 5: Accuracy depending on length respectively task length (see the text for a definition) for different architectures. Data from the experiments with $p = 6/16$ and $n = 4$.

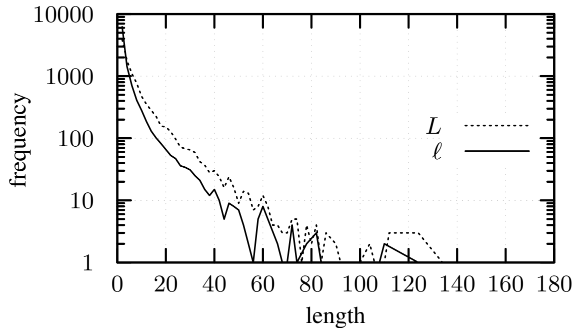


Figure 6: Histogram for length and task length for the experiment with $p = 6/16$ and $n = 4$.

part of the distribution is distributed over small values, so the really long words do not play a big role in the statistics. This naturally raises the question how the RNNs perform for those. Fig. 5 reveals that the performance indeed depends on the length of the sequence respectively the task length and that there are huge differences between the architectures. Only the bigger picture can be compared because the test sets differ between the architectures. While the Elman-RNN reaches perfect accuracy for lengths of up to eight symbols, the GRU gets along very well with lengths of up to 20 symbols. After that, the performance breaks in for these networks. Due to the low number of samples, the curve is very noisy for intermediate values, so it is hard to draw conclusions for this region. There is not a single correct guess by the Elman-RNN for task lengths beyond 90. The LSTM once again performs best in this task and exhibits an almost perfect accuracy over the whole spectrum of lengths.

Finally, the kind of error that is made is of interest. A good representation of that is the confusion matrix given in Fig. 7. For our task, the true bracket is always a closing one. Interestingly, the SRNN appears to “understand” that and hardly ever chooses an opening one. Apart from that Fig. 7 reveals that the SRNN does not consider the different types of brackets as equally likely, otherwise the probability mass would be distributed more evenly.

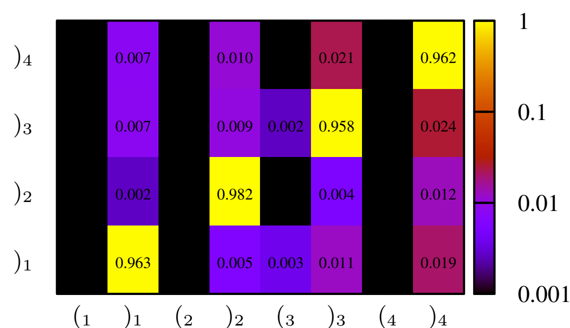


Figure 7: Confusion matrix: Probability of confusing the correct brackets on the y-axis with those on the x-axis, measured with the Elman-RNN for $p = {}^{6}/_{16}$ and $n = 4$. Only non-zero values are given in the plot.

5 Conclusion and Outlook

We evaluated the capability of different RNNs to model an artificial language that consists of convoluted bracket expressions. In terms of perplexity, the models easily get close to the theoretical baseline in most cases. For the task of predicting the last bracket of a sequence, the Elman-RNN mostly reaches accuracies between 0.96 and 1 and hardly ever chooses an opening bracket, while GRU and LSTM score almost perfectly. Based on such good results, our plans for future work are to make the task harder by extending the artificial language. This would help to better carve out the weaknesses of particular architectures. In this context, an important point would be some kind of control over the long-range dependencies.

Acknowledgments

This work was supported in part by the German Research Foundation (DFG) as part of SFB1102. We thank Andrea Fischer for fruitful discussions.

References

Jean-Philippe Bernardy. 2018. Can Recurrent Neural Networks Learn Nested Recursion? *Linguistic Issues in Language Technology*, 16(1).

N. Chomsky. 1956. Three models for the description of language. *IEEE Transactions on Information Theory*, 2(3):113–124.

Kai Lai Chung and William Feller. 1949. On Fluctuations in Coin-tossing. *Proceedings of the National Academy of Sciences*, 35(10):605–608.

Sreerupa Das, C. Lee Giles, and Guo-Zheng Sun. 1992. Learning Context-free Grammars: Capabilities and Limitations of a Recurrent Neural Network with an External Stack Memory. In *Proceedings of The Fourteenth Annual Conference of Cognitive Science Society*, page 14, Bloomington, IN, USA.

Tristan Deleu and Joseph Dureau. 2016. Learning Operations on a Stack with Neural Turing Machines. *Computing Research Repository*, arXiv:1612.00827.

Robert B. Dewell. 2011. *The Meaning of Particle / Prefix Constructions in German*, volume 34 of *Human Cognitive Processing*. John Benjamins Publishing Company.

Philippe Duchon. 2000. On the enumeration and generation of generalized Dyck words. *Discrete Mathematics*, 225(1-3):121–135.

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

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

Yvonne Kalinke and Helko Lehmann. 1998. Computation in recurrent neural networks: From counters to iterated function systems. *Lecture Notes in Computer Science*, 1502:179–190.

Andrej Karpathy, Justin Johnson, and Li Fei-Fei. 2016. Visualizing and Understanding Recurrent Networks. In *International Conference on Learning Representations, Workshop Track*.

Tianyu Li, Guillaume Rabusseau, and Doina Precup. 2017. Neural Network Based Nonlinear Weighted Finite Automata. *Computing Research Repository*, arXiv:1709.04380.

Tal Linzen, Emmanuel Dupoux, and Yoav Goldberg. 2016. Assessing the Ability of LSTMs to Learn Syntax-Sensitive Dependencies. *Transactions of the Association for Computational Linguistics*, 4:521–535.

Tomáš Mikolov, Stefan Kombrink, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2011a. Extensions of Recurrent Neural Network Language Model. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 5528–5531. IEEE.

Tomáš Mikolov, Stefan Kombrink, Anoop Deoras, Lukáš Burget, and Jan Honza Černockỳ. 2011b. RNNLM – Recurrent Neural Network Language Modeling Toolkit. In *ASRU 2011 Demo Session*, Waikoloa, HI, USA.

Peter O. Müller, Ingeborg Ohnheiser, Susan Olsen, and Franz Rainer. 2015. *Word-Formation: An International Handbook of the Languages of Europe*, volume 40.1 of *Handbooks of Linguistics and Communication Science*. De Gruyter Mouton.

Youssef Oualil, Mittul Singh, Clayton Greenberg, and Dietrich Klakow. 2016. Long-short range context neural networks for language modeling. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1473–1481, Austin, Texas.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1985. Learning Internal Representations by Error Propagation. Technical report, University of California San Diego, La Jolla Institute for Cognitive Science.

Xiaoyu Shen, Youssef Oualil, Clayton Greenberg, Mittul Singh, and Dietrich Klakow. 2017. Estimation of gap between current language models and human performance. In *Proceedings of the 18th Annual Conference of the International Speech Communication Association*, pages 553–557, Stockholm, Sweden.

H. S. Sichel. 1974. On a Distribution Representing Sentence-Length in Written Prose. *Journal of the Royal Statistical Society. Series A (General)*, 137(1):25.

Bengt Sigurd, Mats Eeg-Olofsson, and Joost van Weijer. 2004. Word length, sentence length and frequency – Zipf revisited. *Studia Linguistica*, 58(1):37–52.

Gail Weiss, Yoav Goldberg, and Eran Yahav. 2017. Extracting automata from recurrent neural networks using queries and counterexamples. *Computing Research Repository*, arXiv:1711.09576.

Gail Weiss, Yoav Goldberg, and Eran Yahav. 2018. On the practical computational power of finite precision rnns for language recognition. *Computing Research Repository*, arXiv:1805.04908.